

Applied Type System: An Approach to Practical Programming with Theorem-Proving

Hongwei Xi*

Boston University, Boston, MA 02215, USA

(e-mail: hwxi@cs.bu.edu)

Abstract

The framework Pure Type System (**PTS**) offers a simple and general approach to designing and formalizing type systems. However, in the presence of dependent types, there often exist certain acute problems that make it difficult for **PTS** to directly accommodate many common realistic programming features such as general recursion, recursive types, effects (e.g., exceptions, references, input/output), etc. In this paper, Applied Type System (**ATS**) is presented as a framework for designing and formalizing type systems in support of practical programming with advanced types (including dependent types). In particular, it is demonstrated that **ATS** can readily accommodate a paradigm referred to as programming with theorem-proving (PwTP) in which programs and proofs are constructed in a syntactically intertwined manner, yielding a practical approach to internalizing constraint-solving needed during type-checking. The key salient feature of **ATS** lies in a complete separation between statics, where types are formed and reasoned about, and dynamics, where programs are constructed and evaluated. With this separation, it is no longer possible for a program to occur in a type as is otherwise allowed in **PTS**. The paper contains not only a formal development of **ATS** but also some examples taken from **ATS**, a programming language with a type system rooted in **ATS**, in support of employing **ATS** as a framework to formulate advanced type systems for practical programming.

Contents

1 Introduction	1
2 Untyped λ-Calculus λ_{dyn}	7
3 Formal Development of ATS_0	8
4 Formal Development of ATS_{pf}	19
5 Related Work and Conclusion	28
References	29

1 Introduction

A primary motivation for developing Applied Type System (**ATS**) stems from an earlier attempt to support a restricted form of dependent types in practical programming (Xi, 2007).

* Supported in part by NSF grants no. CCR-0224244, no. CCR-0229480, and no. CCF-0702665

While there is already a framework Pure Type System (**PTS**) (Barendregt, 1992) that offers a simple and general approach to designing and formalizing type systems, it is well understood that there often exist some acute problems (in the presence of dependent types) making it difficult for **PTS** to accommodate many common realistic programming features. In particular, various studies reported in the literature indicate that great efforts are often required in order to maintain a style of pure reasoning as is advocated in **PTS** when features such as general recursion (Constable & Smith, 1987), recursive types (Mendler, 1987), effects (Honsell *et al.*, 1995), exceptions (Hayashi & Nakano, 1988) and input/output are present.

The framework **ATS** is formulated to allow for designing and formalizing type systems that can readily support common realistic programming features. The formulation of **ATS** given in this paper is primarily based on the work reported in two previous papers (Xi, 2004; Chen & Xi, 2005) but there are some fundamental changes in terms of the handling of proofs and proof construction. In particular, the requirement is dropped that a proof in **ATS** must be represented as a normalizing lambda-term (Xi, 2008a).

In contrast to **PTS**, the key salient feature of **ATS** lies in a complete separation between statics, where types are formed and reasoned about, from dynamics, where programs are constructed and evaluated. This separation, with its origin in a previous study on a restricted form of dependent types developed in Dependent ML (**DML**) (Xi, 2007), makes it straightforward to support dependent types in the presence of effects such as references and exceptions. Also, with the introduction of two new (and thus somewhat unfamiliar) forms of types: *guarded types* and *asserting types*, **ATS** is able to capture program invariants in a manner that is similar to the use of pre-conditions and post-conditions (Hoare, 1969). By now, studies have shown amply and convincingly that a variety of traditional programming paradigms (e.g., functional programming, object-oriented programming, meta-programming, modular programming) can be directly supported in **ATS** without relying on *ad hoc* extensions, attesting to the expressiveness of **ATS**. In this paper, the primary focus of study is set on a novel programming paradigm referred to as *programming with theorem-proving* (PwTP) and its support in **ATS**. In particular, a type-theoretical foundation for PwTP is to be formally established and its correctness proven.

The notion of type equality plays a pivotal rôle in type system design. However, the importance of this rôle is often less evident in commonly studied type systems. For instance, in the simply typed λ -calculus, two types are considered equal if and only if they are syntactically the same; in the second-order polymorphic λ -calculus (λ_2) (Reynolds, 1972) and System F (Girard, 1986), two types are considered equal if and only if they are α -equivalent; in the higher-order polymorphic λ -calculus (λ_ω), two types are considered equal if and only if they are $\beta\eta$ -equivalent. This situation immediately changes in **ATS**, and let us see a simple example that stresses this point.

In Figure 1, the presented code implements a function in **ATS** (Xi, 2008b), which is a substantial system such that its compiler alone currently consists of more than 165K lines of code implemented in **ATS** itself.¹ The concrete syntax used in the implementation should be accessible to those who are familiar with Standard ML (SML) (Milner *et al.*, 1997)). Note that **ATS** is a programming language equipped with a type system rooted in **ATS**, and

¹ Please see <http://www.ats-lang.org> for more details.

```

fun
append
{a:type}{m,n:nat}
(
  xs: list (a, m), ys: list (a, n)
) : list (a, m+n) =
  case xs of
  | nil() => ys (* the first clause *)
  | cons(x, xs) => cons (x, append(xs, ys)) (* the second clause *)
// end of [append]

```

Fig. 1. List-append in ATS

the name of ATS derives from that of ATS. The type constructor **list** takes two arguments; when applied to a type T and an integer I , **list**(T, I) forms a type for lists of length I in which each element is of type T . Also, the two list constructors **nil** and **cons** are assigned the following types:

$$\begin{aligned}
 \text{nil} & : \forall a : \text{type}. () \rightarrow \text{list}(a, 0) \\
 \text{cons} & : \forall a : \text{type}. \forall n : \text{nat}. (a, \text{list}(a, n)) \rightarrow \text{list}(a, n + 1)
 \end{aligned}$$

So **nil** constructs a list of length 0, and **cons** takes an element and a list of length n to form a list of length $n + 1$. The header of the function **append** indicates that **append** is assigned the following type:

$$\forall a : \text{type}. \forall m : \text{nat}. \forall n : \text{nat}. (\text{list}(a, m), \text{list}(a, n)) \rightarrow \text{list}(a, m + n)$$

which simply means that **append** returns a list of length $m + n$ when applied to one list of length m and another list of length n . Note that *type* is a built-in sort in ATS, and a static term of the sort *type* stands for a type (for dynamic terms). Also, *int* is a built-in sort for integers in ATS, and *nat* is the subset sort $\{a : \text{int} \mid a \geq 0\}$ for all nonnegative integers.

When the above implementation of **append** is type-checked, the following two constraints are generated:

1. $\forall m : \text{nat}. \forall n : \text{nat}. m = 0 \supset n = m + n$
2. $\forall m : \text{nat}. \forall n : \text{nat}. \forall m' : \text{nat}. m = m' + 1 \supset (m' + n) + 1 = m + n$

The first constraint is generated when the first clause is type-checked, which is needed for determining whether the types **list**(a, n) and **list**($a, m + n$) are equal under the assumption that **list**(a, m) equals **list**($a, 0$). Similarly, the second constraint is generated when the second clause is type-checked, which is needed for determining whether the types **list**($a, (m' + n) + 1$) and **list**($a, m + n$) are equal under the assumption that **list**(a, m) equals **list**($a, m' + 1$). Clearly, certain restrictions need to be imposed on the form of constraints allowed in practice so that an effective approach can be found to perform constraint-solving. In DML, a programming language based on DML (Xi, 2007), the constraints generated during type-checking are required to be linear inequalities on integers so that the problem of constraint satisfaction can be turned into the problem of linear integer programming, for which there are many highly practical solvers (albeit the problem of linear integer programming itself is NP-complete). This is indeed a very simple design, but

it can also be too restrictive, sometimes, as nonlinear constraints (e.g., $\forall n : \text{int}. n * n \geq 0$) are commonly encountered in practice. Furthermore, the very nature of such a design indicates its being inherently *ad hoc*.

By combining programming with theorem-proving, a fundamentally different design of constraint-solving can provide the programmer with an option to handle nonlinear constraints through explicit proof construction. For the sake of a simpler presentation, let us assume for this moment that even the addition function on integers cannot appear in the constraints generated during type-checking. Under such a restriction, it is still possible to implement a list-append function in ATS that is assigned a type capturing the invariant that the length of the concatenation of two given lists xs and ys equals $m + n$ if xs and ys are of length m and n , respectively. Let us first see such an implementation given in Figure 2, which is presented here as a motivating example for programming with theorem-proving (PwTP).

The datatypes **Z** and **S** are declared in Figure 2 solely for representing natural numbers: **Z** represents 0, and **S**(N) represents the successor of the natural number represented by N . The data constructors associated with **Z** and **S** are of no use. Given a type T and another type N , **mylist**(T, N) is a type for lists containing n elements of the type T , where n is the natural number represented by N . Note that **mylist** is not a standard datatype (as is supported in ML); it is a *guarded recursive datatype* (GRDT) (Xi *et al.*, 2003), which is also known as *generalized algebraic datatype* (GADT) (Cheney & Hinze, 2003) in Haskell and OCaml. The datatype **addrel** is declared to capture the relation induced by the addition function on natural numbers. Given types M , N , and R representing natural numbers m , n , and r , respectively, the type **addrel**(M, N, R) is for a value representing some proof of $m + n = r$. Note that **addrel** is also a GRDT or GADT. There are two constructors **addrel.z** and **addrel.s** associated with **addrel**, which encode the following two rules:

$$\begin{aligned} 0 + n &= n && \text{for every natural number } n \\ (m + 1) + n &= (m + n) + 1 && \text{for every pair of natural numbers } m \text{ and } n \end{aligned}$$

Let us now take a look at the implementation of **myappend**. Formally, the type assigned to **myappend** can be written as follows:

$$\begin{aligned} &\forall a : \text{type}. \forall m : \text{type}. \forall n : \text{type}. \\ &(\text{mylist}(a, m), \text{mylist}(a, n)) \rightarrow \exists r : \text{type}. (\text{addrel}(m, n, r), \text{mylist}(a, r)) \end{aligned}$$

In essence, this type states the following: Given two lists of length m and n , **myappend** returns a pair such that the first component of the pair is a proof showing that $m + n$ equals r for some natural number r and the second component is a list of length r .

Unlike **append**, type-checking **myappend** does not generate any linear constraints on integers. As a matter of fact, **myappend** can be readily implemented in both Haskell and OCaml (extended with support for generalized algebraic datatypes), where there is no built-in support for handling linear constraints on integers. This is an example of great significance in the sense that it demonstrates concretely an approach to allowing the programmer to write code of the nature of theorem-proving so as to simplify or even eliminate certain constraints that need otherwise to be solved directly during type-checking. With this approach, constraint-solving is effectively internalized, and the programmer can ac-

```

datatype Z() = Z of ()
datatype S(a:type) = S of a
//
datatype
mylist(type, type) =
  | {a:type}
    mynil(a, Z())
  | {a:type}{n:type}
    mycons(a, S(n)) of (a, mylist(a, n))
//
datatype
addrel(type, type, type) =
  | {n:type}
    addrel_z(Z(), n, n) of ()
  | {m,n:type}{r:type}
    addrel_s(S(m), n, S(r)) of addrel(m, n, r)
//
fun
myappend
{a:type}
{m,n:type}
(
  xs: mylist(a, m)
, ys: mylist(a, n)
) : [r:type]
(
  addrel(m, n, r), mylist(a, r)
) =
(
  case xs of
  | mynil() => let
    val pf = addrel_z() in (pf, ys)
  end // end of [mynil]
  | mycons(x, xs) => let
    val (pf, res) = myappend(xs, ys) in (addrel_s(pf), mycons(x, res))
  end // end of [mycons]
)

```

Fig. 2. A motivating example for PwTP in ATS

tively participate in constraint simplification, gaining a tight control in determining what constraints should be passed to the underlying constraint-solver.

There are some major issues with the implementation given in Figure 2. Clearly, representing natural numbers as types is inadequate since there are types that do not represent any natural numbers. More seriously, this representation turns quantification over natural numbers (which is predicative) into quantification over types (which is impredicative), causing unnecessary complications. Also, proof construction (that is, construction of values of types formed by **addrel**) needs to be actually performed at run-time, which causes inefficiency both time-wise and memory-wise. Probably the most important issue is that proof validity is not guaranteed. For instance, it is entirely possible to fake proof construction by making use of non-terminating functions.

```

datasort
mynat = Z of () | S of mynat
//
datatype
mylist(type, mynat) =
  | {a:type}
    mynil(a, Z())
  | {a:type}{n:mynat}
    mycons(a, S(n)) of (a, mylist(a, n))
//
dataprop
addrel(mynat, mynat, mynat) =
  | {y:mynat}
    addrel_z(Z, y, y) of ()
  | {x,y:mynat}{r:mynat}
    addrel_s(S(x), y, S(r)) of addrel(x, y, r)
//
fun
myappend
{a:type}
{m,n:mynat}
(
  xs: mylist(a, m)
, ys: mylist(a, n)
) : [r:mynat]
(
  addrel(m, n, r) | mylist(a, r)
) =
(
  case xs of
  | mynil() => let
    val pf = addrel_z() in (pf | ys)
  end // end of [mynil]
  | mycons(x, xs) => let
    val (pf | res) = myappend(xs, ys) in (addrel_s(pf) | mycons(x, res))
  end // end of [mycons]
)

```

Fig. 3. An example making use of PwTP in ATS

In Figure 3, another implementation of `myappend` is given that makes use of the support for PwTP in ATS. Instead of representing natural numbers as types, a `datasort` of the name `mynat` is declared and natural numbers can be represented as static terms of the sort `mynat`. Also, a `dataprop` **addrel** is declared for capturing the relation induced by the addition function on natural numbers. As a `dataprop`, **addrel** can only form types for values representing proofs, which are erased after type-checking and thus need no construction at run-time. In the implementation of `myappend`, the bar symbol (`|`) is used in place of the comma symbol to separate components in tuples; the components appearing to the left of the bar symbol are proof expressions (to be erased) and those to the right are dynamic expressions (to be evaluated). After proof-erasure, the implementation of `myappend` essentially matches that of `append` given in Figure 1.

As a framework to facilitate the design and formalization of advanced type systems for practical programming, **ATS** is first formulated with no support for PwTP (Xi, 2004). This formulation is the basis for a type system referred to as ATS_0 in this paper. The support for PwTP is added into **ATS** in a subsequent formulation (Chen & Xi, 2005), which serves as the basis for a type system referred to as ATS_{pf} in this paper. However, a fundamentally different approach is adopted in ATS_{pf} to justify the soundness of PwTP, which essentially translates each well-typed program in ATS_{pf} into another well-typed one in ATS_0 of the same dynamic semantics. The identification and formalization of this approach, which is both simpler and more general than one used previously (Chen & Xi, 2005), consists of a major technical contribution of the paper.

It is intended that the paper should focus on the theoretical development of **ATS**, and the presentation given is of a minimalist style. The organization for the rest of the paper is given as follows. An untyped λ -calculus λ_{dyn} is first presented in Section 2 for the purpose of introducing some basic concepts needed to formally assign dynamic (that, operational) semantics to programs. In Section 3, a generic applied type system ATS_0 is formulated and its type-soundness established. Subsequently, ATS_0 is extended to ATS_{pf} in Section 4 with support for PwTP, and the type-soundness of ATS_{pf} is reduced to that of ATS_0 through a translation from well-typed programs in the former to those in the latter. Lastly, some closely related work is discussed in Section 5 and the paper concludes.

2 Untyped λ -Calculus λ_{dyn}

The purpose of formulating λ_{dyn} , an untyped lambda-calculus extended with constants (including constant constructors and constant functions), is to set up some machinery needed to formalize dynamic (that is, operational) semantics for programs. It is to be proven that a well-typed program in **ATS** can be turned into one in λ_{dyn} through type-erasure and proof-erasure while retaining its dynamic semantics, stressing the point that types and proofs in **ATS** play no active rôle in the evaluation of a program. In this regard, the form of typing studied in **ATS** is of Curry-style (in contrast with Church-style) (Reynolds, 1998).

There are no static terms in λ_{dyn} . The syntax for the dynamic terms in λ_{dyn} is given as follows:

$$\begin{aligned} \text{dynamic terms } e ::= & \ x \mid dcx(\vec{e}) \mid \langle e_1, e_2 \rangle \mid \mathbf{fst}(e) \mid \mathbf{snd}(e) \mid \\ & \mathbf{lam } x.e \mid \mathbf{app}(e_1, e_2) \mid \mathbf{let } x = e_1 \mathbf{ in } e_2 \end{aligned}$$

where the notation \vec{e} is for a possibly empty sequence of dynamic terms. Let dcx range over external dynamic constants, which include both dynamic constructors dcc and dynamic functions dcf . The arguments taken by a dynamic constructor or function are often primitive values (instead of those constructed by **lam** and $\langle \cdot, \cdot \rangle$) and the result returned by it is often a primitive value as well. The meaning of various forms of dynamic terms should become clear when the rules for evaluating them are given.

The values in λ_{dyn} are just special forms of dynamic terms, and the syntax for them is given as follows:

$$\text{values } v ::= \ x \mid dcc(\vec{v}) \mid \langle v_1, v_2 \rangle \mid \mathbf{lam } x.e$$

where \vec{v} is for a possibly empty sequence of values. A standard approach to assigning dynamic semantics to terms is based on the notion of evaluation contexts:

$$\text{evaluation contexts } E ::= [] \mid dcx(v_1, \dots, v_{i-1}, E, e_{i+1}, \dots, e_n) \mid \langle E, e \rangle \mid \langle v, E \rangle \mid \mathbf{app}(E, e) \mid \mathbf{app}(v, E) \mid \mathbf{let } x = E \mathbf{ in } e$$

Essentially, an evaluation context E is a dynamic term in which a subterm is replaced with a hole denoted by $[]$. Note that only subterms at certain positions in a dynamic term can be replaced to form valid evaluation contexts.

Definition 2.1

The redexes in λ_{dyn} and their reducts are defined as follows:

- $\mathbf{fst}(\langle v_1, v_2 \rangle)$ is a redex, and its reduct is v_1 .
- $\mathbf{snd}(\langle v_1, v_2 \rangle)$ is a redex, and its reduct is v_2 .
- $\mathbf{app}(\mathbf{lam } x. e, v)$ is a redex, and its reduct is $e[x \mapsto v]$.
- $dcf(\vec{v})$ is a redex if it is defined to equal some value v ; if so, its reduct is v .

Note that it may happen later that a new form of redex can have more than one reducts. Given a dynamic term of the form $E[e_1]$ for some redex e_1 , $E[e_1]$ is said to reduce to $E[e_2]$ in one-step if e_2 is a reduct of e_1 , and this one-step reduction is denoted by $E[e_1] \rightarrow E[e_2]$. Let \rightarrow^* stand for the reflexive and transitive closure of \rightarrow .

Given a program (that is, a closed dynamic term) e_0 in λ_{dyn} , a finite reduction sequence starting from e_0 can either lead to a value or a non-value. If a non-value cannot be further reduced, then the non-value is said to be *stuck* or in a *stuck* form. In practice, values can often be represented in special manners to allow various stuck forms to be detected through checks performed at run-time. For instance, the representation of a value in a dynamically typed language most likely contains a tag to indicate the type of the value. If it is detected that the evaluation of a program reaches a stuck form, then the evaluation can be terminated abnormally with a raised exception.

Detecting potential stuck forms that may occur during the evaluation of a program can also be done statically (that is, at compiler-time). One often imposes a type discipline to ensure the absence of various stuck forms during the evaluation of a well-typed program. This is the line of study to be carried out in the rest of the paper.

3 Formal Development of ATS_0

As a generic applied type system, ATS_0 consists of a static component (statics), where types are formed and reasoned about, and a dynamic component (dynamics), where programs are constructed and evaluated. The statics itself is a simply typed lambda-calculus (extended with certain constants), and the types in it are called *sorts* so as to avoid confusion with the types for classifying dynamic terms, which are themselves static terms.

The syntax for the statics of ATS_0 is given in Figure 4. Let b range over the base sorts in ATS_0 , which include at least *bool* for static booleans and *type* for types (assigned to dynamic terms). The base sort *int* for static integers is not really needed for formalizing ATS_0 but it is often used in the presented examples. Let a and s range over static variables and static terms, respectively. There may be some built-in static constants *scx*, which are either static constant constructors *scc* or static constant functions *scf*. A c-sort is of the

sorts	σ	$::=$	$b \mid \sigma_1 \rightarrow \sigma_2$
static terms	s	$::=$	$a \mid scx(s_1, \dots, s_n) \mid \lambda a : \sigma. s \mid s_1(s_2)$
static var. ctx.	Σ	$::=$	$\emptyset \mid \Sigma, a : \sigma$
static subst.	Θ	$::=$	$\square \mid \Theta[a \mapsto s]$

Fig. 4. The syntax for the statics of ATS_0

form $(\sigma_1, \dots, \sigma_n) \Rightarrow b$, which can only be assigned to static constants. Note that a c-sort is not considered a (regular) sort. Given a static constant scx , a static term $scx(s_1, \dots, s_n)$ is of sort b if scx is assigned a c-sort $(\sigma_1, \dots, \sigma_n) \Rightarrow b$ for some sorts $\sigma_1, \dots, \sigma_n$ and s_i can be assigned the sorts σ_i for $i = 1, \dots, n$. It is allowed to write scc for $scc()$ if there is no risk of confusion. In ATS_0 , the existence of the following static constants with the assigned c-sorts is assumed:

$true$:	$() \Rightarrow bool$
$false$:	$() \Rightarrow bool$
\leq_{ty}	:	$(type, type) \Rightarrow bool$
$*$:	$(type, type) \Rightarrow type$
\rightarrow	:	$(type, type) \Rightarrow type$
\wedge	:	$(bool, type) \Rightarrow type$
\supset	:	$(bool, type) \Rightarrow type$
\forall_σ	:	$(\sigma \rightarrow type) \Rightarrow type$
\exists_σ	:	$(\sigma \rightarrow type) \Rightarrow type$

Note that infix notation may be used for certain static constants. For instance, $s_1 \rightarrow s_2$ stands for $\rightarrow(s_1, s_2)$ and $s_1 \leq_{ty} s_2$ stands for $\leq_{ty}(s_1, s_2)$. In addition, $\forall a : \sigma. s$ and $\exists a : \sigma. s$ stand for $\forall_\sigma(\lambda a : \sigma. s)$ and $\exists_\sigma(\lambda a : \sigma. s)$, respectively. Given a static constant constructor scc , if the c-sort assigned to scc is $(\sigma_1, \dots, \sigma_n) \Rightarrow type$ for some sorts $\sigma_1, \dots, \sigma_n$, then scc is a type constructor. For instance, $*$, \rightarrow , \wedge , \supset , \forall_σ and \exists_σ are all type constructors. Additional built-in base type constructors may be assumed.

Given a proposition B and a type T , $B \supset T$ is a guarded type and $B \wedge T$ is an asserting type. Intuitively, if a value v is assigned a guarded type $B \supset T$, then v can be used only if the guard B is satisfied; if a value v of an asserting type $B \wedge T$ is generated at a program point, then the assertion B holds at that point. For instance, suppose that int is a sort for (static) integers and \mathbf{int} is a type constructor of the sort $(int) \Rightarrow type$; given a static term s of the sort int , $\mathbf{int}(s)$ is a singleton type for the integer equal to s ; hence, the usual type \mathbf{Int} for (dynamic) integers can be defined as $\exists a : int. \mathbf{int}(a)$, and the type \mathbf{Nat} for natural numbers can be defined as $\exists a : int. (a \geq 0) \wedge \mathbf{int}(a)$. Moreover, the following type is for the (dynamic) division function on integers:

$$\forall a_1 : int. \forall a_2 : int. a_2 \neq 0 \supset (\mathbf{int}(a_1), \mathbf{int}(a_2)) \rightarrow \mathbf{int}(a_1/a_2)$$

where the meaning of \neq and $/$ should be obvious. With such a type, division by zero is disallowed during type-checking (at compile-time). Also, suppose that \mathbf{bool} is a type constructor of the sort $(bool) \Rightarrow type$ such that for each proposition B , $\mathbf{bool}(B)$ is a singleton type for the truth value equal to B . Then the usual type \mathbf{Bool} for (dynamic) booleans can be defined as $\exists a : bool. \mathbf{bool}(a)$. The following type is an interesting one:

$$\forall a : bool. \mathbf{bool}(a) \rightarrow a \wedge \mathbf{1}$$

$$\begin{array}{c}
\frac{\Sigma(a) = \sigma}{\Sigma \vdash a : \sigma} \text{ (st-var)} \\
\frac{\vdash scx : (\sigma_1, \dots, \sigma_n) \Rightarrow b \quad \Sigma \vdash s_1 : \sigma_1 \quad \dots \quad \Sigma \vdash s_n : \sigma_n}{\Sigma \vdash scx(s_1, \dots, s_n) : b} \text{ (st-scx)} \\
\frac{\Sigma, a : \sigma_1 \vdash s : \sigma_2}{\Sigma \vdash \lambda a : \sigma_1. s : \sigma_1 \rightarrow \sigma_2} \text{ (st-lam)} \\
\frac{\Sigma \vdash s_1 : \sigma_1 \rightarrow \sigma_2 \quad \Sigma \vdash s_2 : \sigma_1}{\Sigma \vdash s_1(s_2) : \sigma_2} \text{ (st-app)}
\end{array}$$

Fig. 5. The sorting rules for the statics of ATS_0

where **1** stands for the unit type. Given a function f of this type, we can apply f to a boolean value v of type $\mathbf{bool}(B)$ for some proposition B ; if $f(v)$ returns, the B must be true; therefore f acts like dynamic assertion-checking.

For those familiar with qualified types (Jones, 1994), which underlies the type class mechanism in Haskell, it should be noted that a qualified type cannot be regarded as a guarded type. The simple reason is that the proof of a guard in ATS_0 bears no computational significance, that is, it cannot affect the run-time behavior of a program, while a dictionary, which is just a proof of some predicate on types in the setting of qualified types, can and is mostly likely to affect the run-time behavior of a program.

The standard rules for assigning sorts to static terms are given in Figure 5, where the judgement $\vdash scx : (\sigma_1, \dots, \sigma_n) \Rightarrow b$ means that the static constant scx is assumed to be of the c-sort $(\sigma_1, \dots, \sigma_n) \Rightarrow b$. Given $\vec{s} = s_1, \dots, s_n$ and $\vec{\sigma} = \sigma_1, \dots, \sigma_n$, a judgement of the form $\Sigma \vdash \vec{s} : \vec{\sigma}$ means $\Sigma \vdash s_i : \sigma_i$ for $i = 1, \dots, n$. Let B stand for a static term that can be assigned the sort \mathbf{bool} (under some context Σ) and \vec{B} a possibly empty sequence of static boolean terms. Also, let T stand for a type (for dynamic terms), which is a static term that can be assigned the sort \mathbf{type} (under some context Σ). Given contexts Σ_1 and Σ_2 and a substitution Θ , the judgement $\Sigma_1 \vdash \Theta : \Sigma_2$ means that $\Sigma_1 \vdash \Theta(a) : \Sigma_2(a)$ is derivable for each $a \in \mathbf{dom}(\Theta) = \mathbf{dom}(\Sigma_2)$.

Proposition 3.1

Assume $\Sigma \vdash s : \sigma$ is derivable. If $\Sigma = \Sigma_1, \Sigma_2$ and $\Sigma_1 \vdash \Theta : \Sigma_2$ holds, then $\Sigma_1 \vdash s[\Theta] : \sigma$ is derivable.

Proof

By structural induction on the derivation of $\Sigma \vdash s : \sigma$. □

Definition 3.1 (Constraints in ATS_0)

A constraint in ATS_0 is of the form $\Sigma; \vec{B} \models B_0$, where $\Sigma \vdash B : \mathbf{bool}$ holds for each B in \vec{B} and $\Sigma \vdash B_0 : \mathbf{bool}$ holds as well, and the constraint relation in ATS_0 is the one that determines whether each constraint is true or false.

Each regularity rule in Figure 6 is assumed to be met, that is, the conclusion of each regularity rule holds if all of its premisses hold, and the following regularity conditions on \leq_{ty} are also satisfied:

1. $\Sigma; \vec{B} \models T \leq_{ty} T$ holds for every T .

$$\begin{array}{c}
\frac{B \in \vec{B}}{\Sigma; \vec{B} \models B} \text{ (reg-id)} \\
\frac{}{\Sigma; \vec{B} \models \text{true}} \text{ (reg-true)} \\
\frac{}{\Sigma; \vec{B} \models \text{false}} \text{ (reg-false)} \\
\frac{\Sigma; \vec{B} \models B}{\Sigma, a : \sigma; \vec{B} \models B} \text{ (reg-var-thin)} \\
\frac{\Sigma \vdash B_1 : \text{bool} \quad \Sigma; \vec{B} \models B_2}{\Sigma; \vec{B}, B_1 \models B_2} \text{ (reg-bool-thin)} \\
\frac{\Sigma, a : \sigma; \vec{B} \models B \quad \Sigma \vdash s : \sigma}{\Sigma; \vec{B}[a \mapsto s] \models B[a \mapsto s]} \text{ (reg-subst)} \\
\frac{\Sigma; \vec{B} \models B_1 \quad \Sigma; \vec{B}, B_1 \models B_2}{\Sigma; \vec{B} \models B_2} \text{ (reg-cut)}
\end{array}$$

Fig. 6. The regularity rules for the constraint relation in ATS_0

dynamic terms	$e ::=$	$x \mid d\text{cx}\{\vec{s}\}(e_1, \dots, e_n) \mid$ $\langle e_1, e_2 \rangle \mid \text{fst}(e) \mid \text{snd}(e) \mid \text{lam } x. e \mid \text{app}(e_1, e_2) \mid$ $\supset^+(e) \mid \supset^-(e) \mid \text{slam } a. e \mid \text{sapp}(e, s) \mid$ $\wedge(e) \mid \text{let } \wedge(x) = e_1 \text{ in } e_2 \mid \langle s, e \rangle \mid \text{let } \langle a, x \rangle = e_1 \text{ in } e_2$
dynamic values	$v ::=$	$x \mid d\text{cc}\{\vec{s}\}(v_1, \dots, v_n) \mid$ $\langle v_1, v_2 \rangle \mid \text{lam } x. e \mid \supset^+(e) \mid \text{slam } a. e \mid \wedge(v) \mid \langle s, v \rangle$
dynamic var. ctx.	$\Delta ::=$	$\emptyset \mid \Delta, x : T$
dynamic subst.	$\Theta ::=$	$[] \mid \Theta[x \mapsto e]$

Fig. 7. The syntax for the dynamics in ATS_0

2. $\Sigma; \vec{B} \models T \leq_{ty} T'$ and $\Sigma; \vec{B} \models T' \leq_{ty} T''$ implies $\Sigma; \vec{B} \models T \leq_{ty} T''$.
3. $\Sigma; \vec{B} \models T_1 * T_2 \leq_{ty} T'_1 * T'_2$ implies $\Sigma; \vec{B} \models T_1 \leq_{ty} T'_1$ and $\Sigma; \vec{B} \models T_2 \leq_{ty} T'_2$.
4. $\Sigma; \vec{B} \models T_1 \rightarrow T_2 \leq_{ty} T'_1 \rightarrow T'_2$ implies $\Sigma; \vec{B} \models T'_1 \leq_{ty} T_1$ and $\Sigma; \vec{B} \models T_2 \leq_{ty} T'_2$.
5. $\Sigma; \vec{B} \models B \wedge T \leq_{ty} B' \wedge T'$ implies $\Sigma; \vec{B}, B \models B'$ and $\Sigma; \vec{B}, B \models T \leq_{ty} T'$.
6. $\Sigma; \vec{B} \models B \supset T \leq_{ty} B' \supset T'$ implies $\Sigma; \vec{B}, B' \models B$ and $\Sigma; \vec{B}, B' \models T \leq_{ty} T'$.
7. $\Sigma; \vec{B} \models \forall a : \sigma. T \leq_{ty} \forall a : \sigma. T'$ implies $\Sigma, a : \sigma; \vec{B} \models T \leq_{ty} T'$.
8. $\Sigma; \vec{B} \models \exists a : \sigma. T \leq_{ty} \exists a : \sigma. T'$ implies $\Sigma, a : \sigma; \vec{B} \models T \leq_{ty} T'$.
9. $\emptyset; \emptyset \models \text{scc}(T_1, \dots, T_n) \leq_{ty} T'$ implies $T' = \text{scc}(T'_1, \dots, T'_n)$ for some T'_1, \dots, T'_n .

The need for these conditions is to become clear when proofs are constructed in the following presentation for formally establishing various meta-properties of ATS_0 . For instance, the last of the above conditions can be invoked to make the claim that $T' \leq_{ty} T_1 \rightarrow T_2$ implies T' being of the form $T'_1 \rightarrow T'_2$. Note that this condition actually implies the consistency of the constraint relation as not every constraint is valid.

Let us now move onto the dynamic component (dynamics) of ATS_0 . The syntax for the dynamics of ATS_0 is given in Figure 7. Let x range over dynamic variables and $d\text{cx}$

dynamic constants, which include both dynamic constant constructors *dcc* and dynamic constant functions *dcf*. Some (unfamiliar) forms of dynamic terms are to be understood when the rules for assigning types to them are presented. Let v range over values, which are dynamic terms of certain special forms, and Δ range over dynamic variable contexts, which assign types to dynamic variables.

During the formal development of ATS_0 , proofs are often constructed by induction on derivations (represented as trees). Given a judgement J , $\mathcal{D} :: J$ means that \mathcal{D} is a derivation of J , that is, the conclusion of \mathcal{D} is J . Given a derivation \mathcal{D} , $ht(\mathcal{D})$ stands for the height of the tree that represents \mathcal{D} .

In ATS_0 , a typing judgement is of the form $\Sigma; \vec{B}; \Delta \vdash e : T$, and the rules for deriving such a judgement are given in Figure 8. Note that certain obvious side conditions associated with some of the typing rules are omitted for the sake of brevity. For instance, the variable a is not allowed to have free occurrences in \vec{B} , Δ , or T when the rule **(ty- \forall -intr)** is applied.

Given $\vec{B} = B_1, \dots, B_n$, $\vec{B} \supset T$ stands for $B_1 \supset (\dots (B_n \supset T) \dots)$. Given $\vec{a} = a_1, \dots, a_n$ and $\vec{\sigma} = \sigma_1, \dots, \sigma_n$, $\forall \vec{a} : \vec{\sigma}$ stands for the sequence of quantifiers: $\forall a : \sigma_1 \dots \forall a : \sigma_n$. A c-type in ATS_0 is of the form $\forall \vec{a} : \vec{\sigma}. \vec{B} \supset (T_1, \dots, T_n) \Rightarrow T$.

The notation $\vdash dcx : \forall \vec{a} : \vec{\sigma}. \vec{B} \supset (T_1, \dots, T_n) \Rightarrow T$ means that dcx is assumed to have the c-type following it; if dcx is a constructor *dcc*, then T is assumed to be constructed by some *scc* and *dcc* is said to be associated with *scc*. For instance, the list constructors and the integer addition and division functions can be given the following c-types:

$$\begin{array}{ll} \text{nil} & : \quad \forall a : \text{type}. \text{list}(a, 0) \\ \text{cons} & : \quad \forall a : \text{type}. \forall n : \text{int}. n \geq 0 \supset (a, \text{list}(a, n)) \rightarrow \text{list}(a, n+1) \\ \text{iadd} & : \quad \forall a_1 : \text{int}. \forall a_2 : \text{int}. (\text{int}(a_1), \text{int}(a_2)) \Rightarrow \text{int}(a_1 + a_2) \\ \text{isub} & : \quad \forall a_1 : \text{int}. \forall a_2 : \text{int}. (\text{int}(a_1), \text{int}(a_2)) \Rightarrow \text{int}(a_1 - a_2) \\ \text{imul} & : \quad \forall a_1 : \text{int}. \forall a_2 : \text{int}. (\text{int}(a_1), \text{int}(a_2)) \Rightarrow \text{int}(a_1 * a_2) \\ \text{idiv} & : \quad \forall a_1 : \text{int}. \forall a_2 : \text{int}. a_2 \neq 0 \supset (\text{int}(a_1), \text{int}(a_2)) \Rightarrow \text{int}(a_1 / a_2) \end{array}$$

where the type constructors **int** and **list** are type constructors of the c-sorts $(\text{int}) \Rightarrow \text{type}$ and $(\text{type}, \text{int}) \Rightarrow \text{type}$, respectively, and $+$, $-$, $*$, and $/$ are static constant functions of the c-sort $(\text{int}, \text{int}) \Rightarrow \text{int}$.

For a technical reason, the rule **(ty-var)** is to be replaced with the following one:

$$\frac{\Delta(x) = T \quad \Sigma; \vec{B} \models T \leq_{\text{ty}} T'}{\Sigma; \vec{B}; \Delta \vdash x : T'} \quad \text{(ty-var')}$$

which combines **(ty-var)** with **(ty-sub)**. This replacement is needed for establishing the following lemma:

Lemma 3.1

Assume $\mathcal{D} :: \Sigma; \vec{B}; \Delta, x : T_1 \vdash e : T_2$ and $\Sigma; \vec{B} \models T'_1 \leq_{\text{ty}} T_1$. Then there is a derivation \mathcal{D}' for the typing judgement $\Sigma; \vec{B}; \Delta, x : T'_1 \vdash e : T_2$ such that $ht(\mathcal{D}') = ht(\mathcal{D})$.

Proof

The proof follows from structural induction on \mathcal{D} immediately. The only interesting case is the one where the last applied rule is **(ty-var')**, and this case can be handled by simply merging two consecutive applications of the rule **(ty-var')** into one (with the help of the regularity condition stating that \leq_{ty} is transitive). \square

$$\begin{array}{c}
\frac{\vdash \Sigma; \vec{B}; \Delta \quad \Delta(x) = T}{\Sigma; \vec{B}; \Delta \vdash x : T} \text{ (ty-var)} \\
\frac{\Sigma; \vec{B}; \Delta \vdash e : T \quad \Sigma; \vec{B} \models T \leq_{ty} T'}{\Sigma; \vec{B}; \Delta \vdash e : T'} \text{ (ty-sub)} \\
\frac{\vdash dcx : \forall \vec{a} : \vec{\sigma}. \vec{B}_0 \supset (T_1, \dots, T_n) \Rightarrow T \quad \Sigma \vdash \vec{s} : \vec{\sigma} \quad \Sigma; \vec{B} \models B[\vec{a} \mapsto \vec{s}] \text{ for each } B \in \vec{B}_0 \quad \Sigma; \vec{B}; \Delta \vdash e_i : T_i[\vec{a} \mapsto \vec{s}] \text{ for } i = 1, \dots, n}{\Sigma; \vec{B}; \Delta \vdash dcx\{\vec{s}\}(e_1, \dots, e_n) : T[\vec{a} \mapsto \vec{s}]} \text{ (ty-dcx)} \\
\frac{\Sigma; \vec{B}; \Delta \vdash e_1 : T_1 \quad \Sigma; \vec{B}; \Delta \vdash e_2 : T_2}{\Sigma; \vec{B}; \Delta \vdash \langle e_1, e_2 \rangle : T_1 * T_2} \text{ (ty-tup)} \\
\frac{\Sigma; \vec{B}; \Delta \vdash e : T_1 * T_2}{\Sigma; \vec{B}; \Delta \vdash \mathbf{fst}(e) : T_1} \text{ (ty-fst)} \quad \frac{\Sigma; \vec{B}; \Delta \vdash e : T_1 * T_2}{\Sigma; \vec{B}; \Delta \vdash \mathbf{snd}(e) : T_2} \text{ (ty-snd)} \\
\frac{\Sigma; \vec{B}; \Delta, x : T_1 \vdash e : T_2}{\Sigma; \vec{B}; \Delta \vdash \mathbf{lam} x. e : T_1 \rightarrow T_2} \text{ (ty-lam)} \\
\frac{\Sigma; \vec{B}; \Delta \vdash e_1 : T_1 \rightarrow T_2 \quad \Sigma; \vec{B}; \Delta \vdash e_2 : T_1}{\Sigma; \vec{B}; \Delta \vdash \mathbf{app}(e_1, e_2) : T_2} \text{ (ty-app)} \\
\frac{\Sigma; \vec{B}; B; \Delta \vdash e : T}{\Sigma; \vec{B}; \Delta \vdash \supset^+(e) : B \supset T} \text{ (ty-}\supset\text{-intr)} \\
\frac{\Sigma; \vec{B}; \Delta \vdash e : B \supset T \quad \Sigma; \vec{B} \models B}{\Sigma; \vec{B}; \Delta \vdash \supset^-(e) : T} \text{ (ty-}\supset\text{-elim)} \\
\frac{\Sigma; \vec{B} \models B \quad \Sigma; \vec{B}; \Delta \vdash e : T}{\Sigma; \vec{B}; \Delta \vdash \wedge(e) : B \wedge T} \text{ (ty-}\wedge\text{-intr)} \\
\frac{\Sigma; \vec{B}; \Delta \vdash e_1 : B \wedge T_1 \quad \Sigma; \vec{B}; B; \Delta, x : T_1 \vdash e_2 : T_2}{\Sigma; \vec{B}; \Delta \vdash \mathbf{let} \wedge(x) = e_1 \mathbf{in} e_2 : T_2} \text{ (ty-}\wedge\text{-elim)} \\
\frac{\Sigma, a : \sigma; \vec{B}; \Delta \vdash e : T}{\Sigma; \vec{B}; \Delta \vdash \mathbf{slam} a. e : \forall a : \sigma. T} \text{ (ty-}\forall\text{-intr)} \\
\frac{\Sigma; \vec{B}; \Delta \vdash e : \forall a : \sigma. T \quad \Sigma \vdash s : \sigma}{\Sigma; \vec{B}; \Delta \vdash \mathbf{sapp}(e, s) : T[a \mapsto s]} \text{ (ty-}\forall\text{-elim)} \\
\frac{\Sigma \vdash s : \sigma \quad \Sigma; \vec{B}; \Delta \vdash e : T[a \mapsto s]}{\Sigma; \vec{B}; \Delta \vdash \langle s, d \rangle : \exists a : \sigma. T} \text{ (ty-}\exists\text{-intr)} \\
\frac{\Sigma; \vec{B}; \Delta \vdash e_1 : \exists a : \sigma. T_1 \quad \Sigma, a : \sigma; \vec{B}; \Delta, x : T_1 \vdash e_2 : T_2}{\Sigma; \vec{B}; \Delta \vdash \mathbf{let} \langle a, x \rangle = e_1 \mathbf{in} e_2 : T_2} \text{ (ty-}\exists\text{-elim)}
\end{array}$$

Fig. 8. The typing rules for the dynamics of ATS_0

Given $\Sigma, \vec{B}, \Delta_1, \Delta_2$ and θ , the judgement $\Sigma; \vec{B}; \Delta_1 \vdash \theta : \Delta_2$ means that the typing judgement $\Sigma; \vec{B}; \Delta_1 \vdash \theta(x) : \Delta_2(x)$ is derivable for each $x \in \mathbf{dom}(\theta) = \mathbf{dom}(\Delta_2)$.

Lemma 3.2 (Substitution in ATS_0)

Assume $\mathcal{D} :: \Sigma; \vec{B}; \Delta \vdash e : T$ in ATS_0 .

1. If $\vec{B} = \vec{B}_1, \vec{B}_2$ and $\Sigma; \vec{B}_1 \models \vec{B}_2$ holds, then $\Sigma; \vec{B}_1; \Delta \vdash e : T$ is also derivable, where $\Sigma; \vec{B}_1 \models \vec{B}_2$ means $\Sigma; \vec{B}_1 \models B$ holds for each $B \in \vec{B}_2$.
2. If $\Sigma = \Sigma_1, \Sigma_2$ and $\Sigma_1 \vdash \Theta : \Sigma_2$ holds, then $\Sigma_1; \vec{B}[\Theta]; \Delta[\Theta] \vdash d[\Theta] : T[\Theta]$ is also derivable.
3. If $\Delta = \Delta_1, \Delta_2$ and $\Sigma; \vec{B}; \Delta_1 \vdash \theta : \Delta_2$ is derivable, then $\Sigma; \vec{B}; \Delta_1 \vdash d[\theta] : T$ is also derivable.

Proof

By structural induction on the derivation \mathcal{D} . □

Lemma 3.3 (Canonical Forms)

Assume $\mathcal{D} :: \emptyset; \emptyset; \emptyset \vdash v : T$. Then the following statements hold:

1. If $T = T_1 * T_2$, then v is of the form $\langle v_1, v_2 \rangle$.
2. If $T = T_1 \rightarrow T_2$, then v is of the form **lam** $x. e$.
3. If $T = B \wedge T_0$, then v is of the form $\wedge(v_0)$.
4. If $T = B \supset T_0$, then v is of the form $\supset^+(e)$.
5. If $T = \forall a : \sigma. T_0$, then v is of the form **slam** $a. e$.
6. If $T = \exists a : \sigma. T_0$, then v is of the form $\langle s, v_0 \rangle$.
7. If $T = scc(\vec{s}_1)$, then v is of the form $dcc\{\vec{s}_2\}(\vec{v})$ for some dcc associated with scc .

Proof

With Definition 3.1, the lemma follows from structural induction on \mathcal{D} . If the last applied rule in \mathcal{D} is **(ty-sub)**, then the proof goes through by invoking the induction hypothesis on the immediate subderivation of \mathcal{D} . Otherwise, the proof follows from a careful inspection of the typing rules in Figure 8. □

In order to assign (call-by-value) dynamic semantics to the dynamic terms in ATS_0 , let us introduce evaluation contexts as follows:

$$\begin{aligned} \text{eval. ctx. } E ::= & \\ & [] \mid dcf\{\vec{s}\}(\vec{v}, E, \vec{e}) \mid \langle E, d \rangle \mid \langle v, E \rangle \mid \mathbf{app}(E, e) \mid \mathbf{app}(v, E) \mid \\ & \supset^-(E) \mid \wedge(E) \mid \mathbf{let} \wedge(x) = E \mathbf{in} e \mid \mathbf{sapp}(E, s) \mid \langle s, E \rangle \mid \mathbf{let} \langle a, x \rangle = E \mathbf{in} e \end{aligned}$$

Definition 3.2

The redexes and their reducts are defined as follows.

- **fst** $(\langle v_1, v_2 \rangle)$ is a redex, and its reduct is v_1 .
- **snd** $(\langle v_1, v_2 \rangle)$ is a redex, and its reduct is v_2 .
- **app** $(\mathbf{lam} \ x. e, v)$ is a redex, and its reduct is $e[x \mapsto v]$.
- $dcf\{\vec{s}\}(\vec{v})$ is a redex if it is defined to equal some value v ; if so, its reduct is v .
- $\supset^-(\supset^+(e))$ is a redex, and its reduct is e .
- **sapp** $(\mathbf{slam} \ a. e, s)$ is a redex, and its reduct is $e[a \mapsto s]$.
- **let** $\wedge(x) = \wedge(v)$ **in** e is a redex, and its reduct is $e[x \mapsto v]$.
- **let** $\langle a, x \rangle = \langle s, v \rangle$ **in** e is a redex, and its reduct is $e[a \mapsto s][x \mapsto v]$.

Given two dynamic terms e_1 and e_2 such that $e_1 = E[e]$ and $e_2 = E[e']$ for some redex e and its reduct e' , e_1 is said to reduce to e_2 in one step and this one-step reduction is denoted by $e_1 \rightarrow e_2$. Let \rightarrow^* stand for the reflexive and transitive closure of \rightarrow .

It is assumed that the type assigned to each dynamic constant function dcf is appropriate, that is, $\emptyset; \emptyset; \emptyset \vdash v : T$ is derivable whenever $\emptyset; \emptyset; \emptyset \vdash dcf\{\vec{s}\}(v_1, \dots, v_n) : T$ is derivable and v is a reduct of $dcf\{\vec{s}\}(v_1, \dots, v_n)$.

Lemma 3.4 (Inversion)

Assume $\mathcal{D} :: \Sigma; \vec{B}; \Delta \vdash e : T$ in ATS_0 .

1. If $e = \langle e_1, e_2 \rangle$, then there exists $\mathcal{D}' :: \Sigma; \vec{B}; \Delta \vdash e : T$ such that $ht(\mathcal{D}') \leq ht(\mathcal{D})$ and the last rule applied in \mathcal{D}' is **(ty-tup)**.
2. If $e = \mathbf{lam} \ x. e_1$, then there exists $\mathcal{D}' :: \Sigma; \vec{B}; \Delta \vdash e : T$ such that $ht(\mathcal{D}') \leq ht(\mathcal{D})$ and the last applied rule in \mathcal{D}' is **(ty-lam)**.
3. If $e = \rhd^+(e_1)$, then there exists $\mathcal{D}' :: \Sigma; \vec{B}; \Delta \vdash e : T$ such that $ht(\mathcal{D}') \leq ht(\mathcal{D})$ and the last rule applied in \mathcal{D}' is **(ty- \rhd -intr)**.
4. If $e = \wedge(e_1)$, then there exists $\mathcal{D}' :: \Sigma; \vec{B}; \Delta \vdash e : T$ such that $ht(\mathcal{D}') \leq ht(\mathcal{D})$ and the last rule applied in \mathcal{D}' is **(ty- \wedge -intr)**.
5. If $e = \mathbf{slam} \ a. e_1$, then there exists $\mathcal{D}' :: \Sigma; \vec{B}; \Delta \vdash e : T$ such that $ht(\mathcal{D}') \leq ht(\mathcal{D})$, and the last rule applied in \mathcal{D}' is **(ty- \forall -intr)**.
6. If $e = \langle s, e_1 \rangle$, then there exists $\mathcal{D}' :: \Sigma; \vec{B}; \Delta \vdash e : T$ such that $ht(\mathcal{D}') \leq ht(\mathcal{D})$, and the last rule applied in \mathcal{D}' is **(ty- \exists -intr)**.

Proof

Let \mathcal{D}' be \mathcal{D} if \mathcal{D} does not end with an application of the rule **(ty-sub)**. Hence, in the rest of the proof, it can be assumed that the last applied rule in \mathcal{D} is **(ty-sub)**, that is, \mathcal{D} is of the following form:

$$\frac{\mathcal{D}_1 :: \Sigma; \vec{B}; \Delta \vdash e : T' \quad \Sigma; \vec{B} \models T' \leq_{ty} T}{\Sigma; \vec{B}; \Delta \vdash e : T} \text{ (ty-sub)}$$

Let us prove (1) by induction on $ht(\mathcal{D})$. By induction hypothesis on \mathcal{D}_1 , there exists a derivation $\mathcal{D}'_1 :: \Sigma; \vec{B}; \Delta \vdash e : T'$ such that $ht(\mathcal{D}'_1) \leq ht(\mathcal{D}_1)$ and the last applied rule in \mathcal{D}'_1 is **(ty-tup)**:

$$\frac{\mathcal{D}'_{21} :: \Sigma; \vec{B}; \Delta \vdash e_1 : T'_1 \quad \mathcal{D}'_{22} :: \Sigma; \vec{B}; \Delta \vdash e_2 : T'_2}{\Sigma; \vec{B}; \Delta \vdash \langle e_1, e_2 \rangle : T'_1 * T'_2} \text{ (ty-tup)}$$

where $T' = T'_1 * T'_2$ and $e = \langle e_1, e_2 \rangle$. By one of the regularity condition, $T = T_1 * T_2$ for some T_1 and T_2 . By another regularity condition, both $\Sigma; \vec{B} \models T'_1 \leq_{ty} T_1$ and $\Sigma; \vec{B} \models T'_2 \leq_{ty} T_2$ hold. By applying **(ty-sub)** to \mathcal{D}'_{21} , one obtains $\mathcal{D}_{21} :: \Sigma; \vec{B}; \Delta \vdash e_1 : T_1$. By applying **(ty-sub)** to \mathcal{D}'_{22} , one obtains $\mathcal{D}_{22} :: \Sigma; \vec{B}; \Delta \vdash e_2 : T_2$. Let \mathcal{D}' be

$$\frac{\mathcal{D}_{21} :: \Sigma; \vec{B}; \Delta \vdash e_1 : T_1 \quad \mathcal{D}_{22} :: \Sigma; \vec{B}; \Delta \vdash e_2 : T_2}{\Sigma; \vec{B}; \Delta \vdash \langle e_1, e_2 \rangle : T_1 * T_2} \text{ (ty-tup)}$$

and the proof for (1) is done since $ht(\mathcal{D}') = 1 + \max(ht(\mathcal{D}_{21}), ht(\mathcal{D}_{22}))$, which equals $1 + 1 + \max(ht(\mathcal{D}'_{21}), ht(\mathcal{D}'_{22})) = 1 + ht(\mathcal{D}'_1) \leq 1 + ht(\mathcal{D}_1) = ht(\mathcal{D})$.

Let us prove (2) by induction on $ht(\mathcal{D})$. By induction hypothesis on \mathcal{D}_1 , there exists a derivation $\mathcal{D}'_1 :: \Sigma; \vec{B}; \Delta \vdash e : T'$ such that $ht(\mathcal{D}'_1) \leq ht(\mathcal{D}_1)$ and the last applied rule in \mathcal{D}'_1 is **(ty-lam)**:

$$\frac{\mathcal{D}'_2 :: \Sigma; \vec{B}; \Delta, x : T'_1 \vdash e_1 : T'_2}{\Sigma; \vec{B}; \Delta \vdash \mathbf{lam} \ x. e_1 : T'_1 \rightarrow T'_2} \text{ (ty-lam)}$$

where $T' = T'_1 \rightarrow T'_2$ and $e = \mathbf{lam} \ x. e_1$. By one of the regularity conditions, $T = T_1 \rightarrow T_2$ for some T_1 and T_2 . By another regularity condition, both $\Sigma; \vec{B} \models T_1 \leq_{ty} T'_1$ and $\Sigma; \vec{B} \models T'_2 \leq_{ty} T_2$ hold. Hence, by Lemma 3.1, there is a derivation $\mathcal{D}_2'' :: \Sigma; \vec{B}; \Delta, x : T_1 \vdash e_1 : T'_2$ such that $ht(\mathcal{D}_2'') = ht(\mathcal{D}_2')$. Let \mathcal{D}' be the following derivation,

$$\frac{\mathcal{D}_2'' :: \Sigma; \vec{B}; \Delta, x : T_1 \vdash e_1 : T'_2 \quad \Sigma; \vec{B} \models T'_2 \leq_{ty} T_2}{\Sigma; \vec{B}; \Delta, x : T_1 \vdash e_1 : T_2} \text{ (ty-sub)}$$

$$\frac{\Sigma; \vec{B}; \Delta, x : T_1 \vdash e_1 : T_2}{\Sigma; \vec{B}; \Delta \vdash \mathbf{lam} \ x. e_1 : T_1 \rightarrow T_2} \text{ (ty-lam)}$$

and the proof for (2) is done since $ht(\mathcal{D}') = 1 + 1 + ht(\mathcal{D}_2'') = 1 + 1 + ht(\mathcal{D}_2') = 1 + ht(\mathcal{D}_1') \leq 1 + ht(\mathcal{D}_1) = ht(\mathcal{D})$.

The rest of statements (3), (4), (5), and (6) can all be proven similarly. \square

Theorem 3.1 (Subject Reduction in ATS₀)

Assume $\mathcal{D} :: \Sigma; \vec{B}; \Delta \vdash e : T$ in ATS₀ and $e \rightarrow e'$ holds. Then $\Sigma; \vec{B}; \Delta \vdash e' : T$ is also derivable in ATS₀.

Proof

The proof proceeds by induction on $ht(\mathcal{D})$.

- The last applied rule in \mathcal{D} is **(ty-sub)**:

$$\frac{\mathcal{D}_1 :: \Sigma; \vec{B}; \Delta \vdash e : T' \quad \Sigma \models T' \leq_{ty} T}{\Sigma; \vec{B}; \Delta \vdash e : T}$$

By induction hypothesis on \mathcal{D}_1 , $\mathcal{D}'_1 :: \Sigma; \vec{B}; \Delta \vdash e' : T'$ is derivable, and thus the following derivation is obtained:

$$\frac{\mathcal{D}'_1 :: \Sigma; \vec{B}; \Delta \vdash e' : T' \quad \Sigma \models T' \leq_{ty} T}{\Sigma; \vec{B}; \Delta \vdash e' : T}$$

- The last applied rule in \mathcal{D} is not **(ty-sub)**. Assume that $e = E[e_0]$ and $e' = E[e'_0]$, where e_0 is a redex and e'_0 is a reduct of e_0 . All the cases where E is not \square can be readily handled, and some details are given as follows on the case where $E = \square$ (that is, e is itself a redex).

— \mathcal{D} is of the following form:

$$\frac{\mathcal{D}_1 :: \Sigma; \vec{B}; \Delta \vdash \langle v_{11}, v_{12} \rangle : T_1 * T_2}{\Sigma; \vec{B}; \Delta \vdash \mathbf{fst}(\langle v_{11}, v_{12} \rangle) : T_1} \text{ (ty-fst)}$$

where $T = T_1$ and $e = \mathbf{fst}(\langle v_{11}, v_{12} \rangle)$. By Lemma 3.4, \mathcal{D}_1 may be assumed to be of the following form:

$$\frac{\mathcal{D}_{21} :: \Sigma; \vec{B}; \Delta \vdash v_{11} : T_1 \quad \mathcal{D}_{22} :: \Sigma; \vec{B}; \Delta \vdash v_{12} : T_2}{\Sigma; \vec{B}; \Delta \vdash \langle v_{11}, v_{12} \rangle : T_1 * T_2} \text{ (ty-tup)}$$

Note that $e' = v_{11}$, and the case concludes.

— \mathcal{D} is of the following form:

$$\frac{\mathcal{D}_1 :: \Sigma; \vec{B}; \Delta \vdash \mathbf{lam} \ x. e_1 : T_1 \rightarrow T_2 \quad \mathcal{D}_2 :: \Sigma; \vec{B}; \Delta \vdash v_2 : T_1}{\Sigma; \vec{B}; \Delta \vdash \mathbf{app}(\mathbf{lam} \ x. e_1, v_2) : T_2} \text{ (ty-app)}$$

where $T = T_2$ and $e = \mathbf{app}(\mathbf{lam} \ x. e_1, v_2)$. By Lemma 3.4, \mathcal{D}_1 may be assumed to be of the following form:

$$\frac{\Sigma; \vec{B}; \Delta, x : T_1 \vdash e_1 : T_2}{\Sigma; \vec{B}; \Delta \vdash \mathbf{lam} \ x. e_1 : T_1 \rightarrow T_2}$$

By Lemma 3.2 (Substitution), $\Sigma; \vec{B}; \Delta \vdash e_1[x \mapsto v_2] : T_2$ is derivable. Note that $e' = e_1[x \mapsto v_2]$, and the case concludes.

All of the other cases can be handled similarly. □

For a less involved presentation, let us assume that any well-typed closed value of the form $dcf\{\vec{s}\}(v_1, \dots, v_n)$ is a redex, that is, the dynamic constant function dcf is well-defined at the arguments v_1, \dots, v_n .

Theorem 3.2 (Progress in ATS_0)

Assume that $\mathcal{D} :: \emptyset; \emptyset; \emptyset \vdash e : T$ in ATS_0 . Then either e is a value or $e \rightarrow e'$ holds for some dynamic term e' .

Proof

With Lemma 3.3 (Canonical Forms), the proof proceeds by a straightforward structural induction on \mathcal{D} . □

By Theorem 3.1 and Theorem 3.2, it is clear that for each closed well-typed dynamic term e , $e \rightarrow^* v$ holds for some value v , or there is an infinite reduction sequence starting from e : $e = e_0 \rightarrow e_1 \rightarrow e_2 \rightarrow \dots$. In other words, the evaluation of a well-typed program in ATS_0 either reaches a value or goes on forever (as it can never get stuck). This meta-property of ATS_0 is often referred to as its type-soundness. Per Robin Milner, a catchy slogan for type-soundness states that *a well-typed program can never go wrong*.

$$\begin{aligned} \|x\| &= x \\ \|dcx\{\vec{s}\}(e_1, \dots, e_n)\| &= dcx(\|e_1\|, \dots, \|e_n\|) \\ \|\mathbf{lam} \ x. e\| &= \mathbf{lam} \ x. \|e\| \\ \|\mathbf{app}(e_1, e_2)\| &= \mathbf{app}(\|e_1\|, \|e_2\|) \\ \|\supset^+(e)\| &= \|e\| \\ \|\supset^-(e)\| &= \|e\| \\ \|\wedge(e)\| &= \|e\| \\ \|\mathbf{let} \ \wedge(x) = e_1 \ \mathbf{in} \ e_2\| &= \mathbf{let} \ x = \|e_1\| \ \mathbf{in} \ \|e_2\| \\ \|\mathbf{slam} \ a. e\| &= \|e\| \\ \|\mathbf{sapp}(e, s)\| &= \|e\| \end{aligned}$$

Fig. 9. The type-erasure function $\|\cdot\|$ on dynamic terms

After a program in ATS passes type-checking, it goes through a process referred to as type-erasure to have the static terms inside it completely erased. In Figure 9, a function performing type-erasure is defined, which maps each dynamic term in ATS_0 to an untyped dynamic term in λ_{dyn} .

In order to guarantee that a value in ATS_0 is mapped to another value in λ_{dyn} by the function $\|\cdot\|$, the following syntactic restriction is needed:

- Only when e is a value can the dynamic term $\supset^+(e)$ be formed.
- Only when e is a value can the dynamic term **slam** $a.e$ be formed.

This kind of restriction is often referred to as value-form restriction.

Proposition 3.2

With the value-form restriction being imposed, $\|v\|$ is a value in λ_{dyn} for every value v in ATS_0 .

Proof

By structural induction on v . □

Note that it is certainly possible to have a non-value e in ATS_0 whose type-erasure is a value in λ_{dyn} . From this point on, the value-form restriction is always assumed to have been imposed when type-erasure is performed.

Proposition 3.3

Assume that e_1 is a well-typed closed dynamic term in ATS_0 . If $e_1 \rightarrow e_2$ holds, then either $\|e_1\| = \|e_2\|$ or $\|e_1\| \rightarrow \|e_2\|$ holds in λ_{dyn} .

Proof

By a careful inspection of the forms of redexes in Definition 3.2. □

Proposition 3.4

Assume that e_1 is a well-typed closed dynamic term in ATS_0 . If $\|e_1\| \rightarrow e'_2$ holds in λ_{dyn} , then there exists e_2 such that $e_1 \rightarrow^* e_2$ holds in ATS_0 and $\|e_2\| = e'_2$.

Proof

By induction on the height of the typing derivation for e_1 . □

By Proposition 3.3 and Proposition 3.4, it is clear that type-erasure cannot alter the dynamic semantics of a well-typed dynamic term in ATS_0 .

The formulation of ATS_0 presented in this section is of a minimalist style. In particular, the constraint relation in ATS_0 is treated abstractly. In practice, if a concrete instance of ATS_0 is to be implemented, then rules need to be provided for simplifying constraints. For instance, the following rule may be present:

$$\frac{\Sigma; \vec{B} \models I_1 = I_2}{\Sigma; \vec{B} \models \mathbf{int}(I_1) \leq_{ty} \mathbf{int}(I_2)}$$

With this rule, $\mathbf{int}(I_1) \leq_{ty} \mathbf{int}(I_2)$ can be simplified to the constraint $I_1 = I_2$, where the equality is on static integer terms. The following rule may also be present:

$$\frac{\Sigma; \vec{B} \models T_1 \leq_{ty} T_2 \quad \Sigma; \vec{B} \models I_1 = I_2}{\Sigma; \vec{B} \models \mathbf{list}(T_1, I_1) \leq_{ty} \mathbf{list}(T_2, I_2)}$$

With this rule, $\mathbf{list}(T_1, I_1) \leq_{ty} \mathbf{list}(T_2, I_2)$ can be simplified to the two constraints $T_1 \leq_{ty} T_2$ and $I_1 = I_2$.

For those interested in implementing an applied type system, please find more details in a paper on **DML** (Xi, 2007), which is regarded a special kind of applied type system.

4 Formal Development of ATS_{pf}

Let us extend ATS_0 to ATS_{pf} in this section with support for programming with theorem-proving (PwTP).

A great limitation on employing ATS_0 as the basis for a practical programming language lies in the very rigid handling of constraint-solving in ATS_0 . One is often forced to impose various *ad hoc* restrictions on the syntactic form of a constraint that can actually be supported in practice (so as to match the capability of the underlying constraint-solver), greatly diminishing the effectiveness of using types to capture programming invariants. For instance, only quantifier-free constraints that can be translated into problems of linear integer programming are allowed in the DML programming language (Xi, 2001).

With PwTP being supported in a programming language, programming and theorem-proving can be combined in a syntactically intertwined manner (Chen & Xi, 2005); if a constraint cannot be handled directly by the underlying constraint-solver, then it is possible to simplify the constraint or even eliminate it through explicit proof construction. PwTP advocates an open style of constraint-solving by providing a means within the programming language itself to allow the programmer to actively participate in constraint-solving. In other words, PwTP can be viewed as a programming paradigm for internalizing constraint-solving.

\leq_{pr}	:	$(prop, prop) \Rightarrow bool$
$*$:	$(prop, prop) \Rightarrow prop$
$*$:	$(prop, type) \Rightarrow type$
\rightarrow	:	$(prop, prop) \Rightarrow prop$
\rightarrow	:	$(prop, type) \Rightarrow type$
\wedge	:	$(bool, prop) \Rightarrow prop$
\supset	:	$(bool, prop) \Rightarrow prop$
\forall_{σ}	:	$(\sigma \rightarrow prop) \Rightarrow prop$
\exists_{σ}	:	$(\sigma \rightarrow prop) \Rightarrow prop$

Fig. 10. Additional static constants in ATS_{pf}

Let us now start with the formulation of ATS_{pf} , which extends that of ATS_0 fairly lightly. In addition to the base sorts in ATS_0 , ATS_{pf} contains another base sort *prop*, which is for static terms representing types for proofs. A static term of the sort *prop* may be referred to as a prop (or, sometimes, a type for proofs). Also, it is assumed that the static constants listed in Figure 10 are included in ATS_{pf} . Note that the symbols referring to these static constants may be overloaded. In the following representation, *P* stands for a prop, *T* stands for a type, and *T** stands for either a prop or a type.

The syntax for dynamic terms in ATS_{pf} is essentially the same as that in ATS_0 but with a few minor changes to be mentioned as follows. Some dynamic constructs in ATS_0 need to be split when they are incorporated into ATS_{pf} . The construct $\langle e_1, e_2 \rangle$ for forming tuples is split into $\langle e_1, e_2 \rangle_{pp}$, $\langle e_1, e_2 \rangle_{pt}$, and $\langle e_1, e_2 \rangle_{tt}$ for prop-type pairs, prop-type pairs and type-type pairs, respectively. For instance, a prop-type pair is one where the first component is assigned a prop and the second one a type. Note that there are no type-prop pairs. The construct **lam** *x*. *e* for forming lambda-abstractions is split into **lam**_{pp} *x*. *e*, **lam**_{pt} *x*. *e*, and **lam**_{tt} *x*. *e* for prop-prop functions, prop-type functions and type-type func-

tions, respectively. For instance, a prop-type function is one where the argument is assigned a prop and the body a type. The construct $\mathbf{app}(e_1, e_2)$ for forming applications is split into $\mathbf{app}_{pp}(e_1, e_2)$, $\mathbf{app}_{tp}(e_1, e_2)$, and $\mathbf{app}_{tt}(e_1, e_2)$ for prop-prop applications, type-prop applications and type-type applications. For instance, a type-prop application is one where the function part is assigned a type and the argument a prop. Note that there are no type-prop functions.

The dynamic variable contexts in ATS_{pf} are defined as follows:

$$\text{dynamic var. ctx. } \Delta ::= \emptyset \mid \Delta, x : T^*$$

The regularity conditions on \leq_{ty} needs to be extended with the following two for the new forms of types:

$$3.2 \ \Sigma; \vec{B} \models P_1 * T_2 \leq_{ty} P'_1 * T'_2 \text{ implies } \Sigma; \vec{B} \models P_1 \leq_{pr} P'_1 \text{ and } \Sigma; \vec{B} \models T_2 \leq_{ty} T'_2.$$

$$4.2 \ \Sigma; \vec{B} \models P_1 \rightarrow T_2 \leq_{ty} P'_1 \rightarrow T'_2 \text{ implies } \Sigma; \vec{B} \models P'_1 \leq_{pr} P_1 \text{ and } \Sigma; \vec{B} \models T_2 \leq_{ty} T'_2.$$

It should be noted that there are no regularity conditions imposed on props (as it is not expected for proofs to have any computational meaning).

There are two kinds of typing rules in ATS_{pf} : p-typing rules and t-typing rules, where the former is for assigning props to dynamic terms (encoding proofs) and the latter for assigning types to dynamic terms (to be evaluated). The typing rules for ATS_{pf} are essentially those for ATS_0 listed in Figure 8 except for the following changes:

- Each occurrence of T in the rules for ATS_0 needs to be replaced with T^* .
- The premisses of each p-typing rule (that is, one for assigning a prop to a dynamic term) are required to be p-typing rules themselves.

As an example, let us take a look at the following rule:

$$\frac{\Sigma; \vec{B}; \Delta \vdash e : T \quad \Sigma; \vec{B} \models T \leq_{ty} T'}{\Sigma; \vec{B}; \Delta \vdash e : T'} \text{ (ty-sub)}$$

which yields the following two valid versions:

$$\frac{\Sigma; \vec{B}; \Delta \vdash e : P \quad \Sigma; \vec{B} \models P \leq_{pr} P'}{\Sigma; \vec{B}; \Delta \vdash e : P'} \text{ (ty-sub-p)}$$

$$\frac{\Sigma; \vec{B}; \Delta \vdash e : T \quad \Sigma; \vec{B} \models T \leq_{ty} T'}{\Sigma; \vec{B}; \Delta \vdash e : T'} \text{ (ty-sub-t)}$$

As another example, let us take a look at the following rule:

$$\frac{\Sigma; \vec{B}; \Delta \vdash e : T_1 * T_2}{\Sigma; \vec{B}; \Delta \vdash \mathbf{fst}(e) : T_1} \text{ (ty-fst)}$$

which yields the following two valid versions:

$$\frac{\Sigma; \vec{B}; \Delta \vdash e : P_1 * P_2}{\Sigma; \vec{B}; \Delta \vdash \mathbf{fst}(e) : P_1} \text{ (ty-fst-pp)}$$

$$\frac{\Sigma; \vec{B}; \Delta \vdash e : T_1 * T_2}{\Sigma; \vec{B}; \Delta \vdash \mathbf{fst}(e) : T_1} \text{ (ty-fst-tt)}$$

Note that there is no type of the form $T_1 * T_2$ (for the sake of simplicity). The following version is invalid:

$$\frac{\Sigma; \vec{B}; \Delta \vdash e : P_1 * T_2}{\Sigma; \vec{B}; \Delta \vdash \mathbf{fst}(e) : P_1} \text{ (ty-fst-pt)}$$

because a p-typing rule cannot have any t-typing rule as its premise. Instead, the following typing rule is introduced as the elimination rule for $P_1 * T_2$:

$$\frac{\Sigma; \vec{B}; \Delta \vdash e : P_1 * T_2 \quad \Sigma; \vec{B}; \Delta, x_1 : P_1, x_2 : T_2 \vdash e_0 : T_0}{\Sigma; \vec{B}; \Delta \vdash \mathbf{let} \langle x_1, x_2 \rangle_{pt} = e \mathbf{ in } e_0 : T_0} \text{ (ty-*elim-pt)}$$

As yet another example, let us take a look at the following rule:

$$\frac{\Sigma; \vec{B}; \Delta \vdash e_1 : T_1^* \rightarrow T_2^* \quad \Sigma; \vec{B}; \Delta \vdash e_2 : T_1^*}{\Sigma; \vec{B}; \Delta \vdash \mathbf{app}(e_1, e_2) : T_2^*} \text{ (ty-app)}$$

which yields the following three versions:

$$\begin{aligned} & \frac{\Sigma; \vec{B}; \Delta \vdash e_1 : P_1 \rightarrow P_2 \quad \Sigma; \vec{B}; \Delta \vdash e_2 : P_1}{\Sigma; \vec{B}; \Delta \vdash \mathbf{app}_{pp}(e_1, e_2) : P_2} \text{ (ty-app-pp)} \\ & \frac{\Sigma; \vec{B}; \Delta \vdash e_1 : P_1 \rightarrow T_2 \quad \Sigma; \vec{B}; \Delta \vdash e_2 : P_1}{\Sigma; \vec{B}; \Delta \vdash \mathbf{app}_{tp}(e_1, e_2) : T_2} \text{ (ty-app-tp)} \\ & \frac{\Sigma; \vec{B}; \Delta \vdash e_1 : T_1 \rightarrow T_2 \quad \Sigma; \vec{B}; \Delta \vdash e_2 : T_1}{\Sigma; \vec{B}; \Delta \vdash \mathbf{app}_{tt}(e_1, e_2) : T_2} \text{ (ty-app-tt)} \end{aligned}$$

The first one is a p-typing rule while the other two are t-typing rules.

In ATS_{pf} , the two sorts *bool* and *prop* are intimately related but are also fundamentally different. Gaining a solid understanding of the relation between these two is the key to understanding the design of ATS_{pf} . One may see *prop* as an internalized version of *bool*. Given a static boolean term *B*, its truth value is determined by a constraint-solver outside ATS_{pf} . Given a static term *P* of the sort *prop*, a proof of *P* can be constructed inside ATS_{pf} to attest to the validity of the boolean term encoded by *P*. For clarification, let us see a simple example illustrating the relation between *bool* and *prop* in concrete terms.

```
dataprop
fact_p(int, int) =
| fact_p_bas(0, 1) of ()
| {n:nat}{r:int}
  fact_p_ind(n+1, (n+1)*r) of fact_p(n, r)
```

Fig. 11. A dataprop for encoding the factorial function

In Figure 11, the dataprop *fact_p* declared in ATS is associated with two proof constructors that are assigned the following c-types (or, more precisely, c-props):

$$\begin{aligned} \mathbf{fact_p_bas} & : \text{fact_p}(0, 1) \\ \mathbf{fact_p_ind} & : \forall n : \text{nat}. \forall r : \text{int}. (\text{fact_p}(n, r)) \Rightarrow \text{fact_p}(n + 1, (n + 1) * r) \end{aligned}$$

Let *fact*(*n*) be the value of the factorial function on *n*, where *n* ranges over natural numbers. Given a natural number *n* and an integer *r*, the prop *fact_p*(*n*, *r*) encodes the relation

```

stacst
fact_b : (int, int) -> bool
praxi
fact_b_bas
(
  // argless
) : [fact_b(0, 1)] unit_p
praxi
fact_b_ind{n:int}{r:int}
(
  // argless
) : [n >= 0 && fact_b(n, r) ->> fact_b(n+1, (n+1)*r)] unit_p

```

Fig. 12. A static predicate and two associated proof functions

$fact(n) = r$. In other words, if a proof of the prop $fact_p(n, r)$ can be constructed, then $fact(n)$ equals r .

In Figure 12, a static predicate $fact_b$ is introduced, which corresponds to $fact_p$. Given a natural number n and an integer r , $fact_b(n, r)$ simply means $fact(n) = r$. The two proof functions $fact_b_bas$ and $fact_b_ind$ are assigned the following c-props:

$$\begin{aligned}
 fact_b_bas : () &\Rightarrow fact_b(0, 1) \wedge \mathbf{1} \\
 fact_b_ind : \forall n : int. \forall r : int. () &\Rightarrow (n \geq 0 \wedge fact_b(n, r) \supset fact_b(n+1, (n+1) \cdot r)) \wedge \mathbf{1}
 \end{aligned}$$

where $\mathbf{1}$ is the unit prop (instead of the unit type) that encodes the static truth value *true*. Note that the keyword *praxi* in ATS is used to introduce proof functions that are treated as axioms.

In Figure 13, a verified implementation of the factorial function is given in ATS. Given a natural numbers n , f_fact_p returns an integer r paired with a proof of $fact_p(n, r)$ that attests to the validity of $fact(n) = r$. Note that this implementation makes explicit use of proofs. The constraints generated from type-checking the code in Figure 13 are quantifier-free, and they can be readily solved by the built-in constraint-solver (based on linear integer programming) for ATS.

In Figure 14, another verified implementation of the factorial function is given in ATS. Given a natural numbers, f_fact_b returns an integer r plus the assertion $fact_b(n, r)$ that states $fact(n) = r$. This implementation does not make explicit use of proofs. Applying the keyword *\$solver_assert* to a proof turns the prop of the proof into a static boolean term (of the same meaning) and then adds the term as an assumption to be used for solving the constraints generated subsequently in the same scope. For instance, the two applications of *\$solver_assert* essentially add the following two assumptions:

$$\begin{aligned}
 &fact_b(0, 1) \\
 &\forall n : int. \forall r : int. n \geq 0 \wedge fact_b(n, r) \supset fact_b(n+1, (n+1) \cdot r)
 \end{aligned}$$

Note that the second assumption is universally quantified. In general, solving constraints involving quantifiers is much more difficult than those that are quantifier-free. For instance, the constraints generated from type-checking the code in Figure 14 cannot be solved by the built-in constraint-solver for ATS. Instead, these constraints need to be

```

fun
f_fact_p
  {n:nat}
  (
    n: int(n)
  ) : [r:int]
    (fact_p(n, r) | int(r)) = let
//
fun
loop
{ i:nat
| i <= n
} {r:int}
(
  pf: fact_p(i, r)
| i: int(i), r: int(r)
) : [r:int] (fact_p(n, r) | int(r)) =
  if i < n then
    loop(fact_p_ind(pf) | i+1, (i+1)*r) else (pf | r)
  // end of [if]
//
in
  loop(fact_p_bas() | 0(*i*), 1(*r*))
end // end of [f_fact_p]

```

Fig. 13. A verified implementation of the factorial function

```

fun
f_fact_b
  {n:nat}
  (
    n: int(n)
  ) : [r:int]
    (fact_b(n, r) && int(r)) = let
//
prval() = $solver_assert(fact_b_bas)
prval() = $solver_assert(fact_b_ind)
//
fun
loop
{ i:nat | i <= n }
{ r:int | fact_b(i, r) }
(
  i: int(i), r: int(r)
) : [r:int]
  (fact_b(n, r) && int(r)) =
  if i < n then loop(i+1, (i+1)*r) else (r)
//
in
  loop(0, 1)
end // end of [f_fact_b]

```

Fig. 14. Another verified implementation of the factorial function

exported so that external constraint-solvers (for instance, one based on the Z3 theorem-prover (de Moura & Bjørner, 2008)) can be invoked to solve them.

By comparing these two verified implementations of the factorial function, one sees a concrete case where PwTP (as is done in Figure 13) is employed to simplify the constraints generated from type-checking. This kind of constraint simplification through PwTP is a form of internalization of constraint-solving, and it can often play a pivotal rôle in practice, especially, when there is no effective method available for solving general unsimplified constraints.

Instead of assigning (call-by-value) dynamic semantics to the dynamic terms in ATS_{pf} directly, a translation often referred to as proof-erasure is to be defined that turns each dynamic term in ATS_{pf} into one in ATS_0 of the same dynamic semantics.

Given a sort σ , its proof-erasure $|\sigma|$ is the one in which every occurrence of *prop* in σ is replaced with *bool*.

Given a static variable context Σ , its proof-erasure $|\Sigma|$ is obtained from replacing each declaration $a : \sigma$ with $a : |\sigma|$.

For every static constant scx of the c-sort $(\sigma_1, \dots, \sigma_n) \Rightarrow \sigma$, it is assumed that there exists a corresponding scx' of the c-sort $(|\sigma_1|, \dots, |\sigma_n|) \Rightarrow |\sigma|$; this corresponding scx' may be denoted by $|scx|$. Note that it is possible to have $|scx_1| = |scx_2|$ for different constants scx_1 and scx_2 .

Let us assume the existence of the following static constants:

$$\begin{aligned} \wedge & : (bool, bool) \Rightarrow bool \\ \supset & : (bool, bool) \Rightarrow bool \\ \forall_\sigma & : (\sigma \rightarrow bool) \Rightarrow bool \\ \exists_\sigma & : (\sigma \rightarrow bool) \Rightarrow bool \end{aligned}$$

Note that the symbols referring to these static constants are all overloaded. Naturally, \wedge and \supset are interpreted as the boolean conjunction and boolean implication, respectively, and \forall_σ and \exists_σ are interpreted as the standard universal quantification and existential quantification, respectively. For instance, some pairs of corresponding static constants are listed as follows:

- The boolean implication function \supset corresponds to the prop predicate \leq_{pr} .
- The boolean implication function \supset corresponds to the prop constructor \rightarrow of the c-sort $(prop, prop) \Rightarrow prop$.
- The boolean implication function \supset corresponds to the prop constructor \supset of the c-sort $(bool, prop) \Rightarrow prop$.
- The boolean conjunction function \wedge corresponds to the prop constructor $*$ of the c-sort $(prop, prop) \Rightarrow prop$.
- The boolean conjunction function \wedge corresponds to the prop constructor \wedge of the c-sort $(bool, prop) \Rightarrow prop$.
- The type constructor \wedge of the c-sort $(bool, type) \Rightarrow type$ corresponds to the type constructor $*$ of the c-sort $(prop, type) \Rightarrow type$.
- The type constructor \supset of the c-sort $(bool, type) \Rightarrow type$ corresponds to the type constructor \rightarrow of the c-sort $(prop, type) \Rightarrow type$.
- For each sort σ , the universal quantifier \forall_σ of the sort $(\sigma \rightarrow bool) \Rightarrow bool$ corresponds to the universal quantifier \forall_σ of the sort $(\sigma \rightarrow prop) \Rightarrow prop$.

$$\begin{aligned}
|x| &= x \\
|dcx\{\vec{s}\}(\vec{e})| &= dcx\{|\vec{s}|\}(|\vec{e}|) \\
|\langle e_1, e_2 \rangle_{pt}| &= \wedge(|e_2|) \\
|\langle e_1, e_2 \rangle_{tt}| &= \langle |e_1|, |e_2| \rangle_{tt} \\
|\mathbf{fst}(e)| &= \mathbf{fst}(|e|) \\
|\mathbf{snd}(e)| &= \mathbf{snd}(|e|) \\
|\mathbf{let} \langle x_p, x_t \rangle_{pt} = e_1 \text{ in } e_2| &= \mathbf{let} \wedge(x_t) = |e_1| \text{ in } |e_2| \\
|\mathbf{lam}_{pt} x. e| &= \supset^+(|e|) \\
|\mathbf{lam}_{tt} x. e| &= \mathbf{lam} x. |e| \\
|\mathbf{app}_{tp}(e_1, e_2)| &= \supset^-(|e_1|) \\
|\mathbf{app}_{tt}(e_1, e_2)| &= \mathbf{app}(|e_1|, |e_2|) \\
|\supset^+(e)| &= \supset^+(|e|) \\
|\supset^-(e)| &= \supset^-(|e|) \\
|\wedge(e)| &= \wedge(|e|) \\
|\mathbf{let} \wedge(x) = e_1 \text{ in } e_2| &= \mathbf{let} \wedge(x) = |e_1| \text{ in } |e_2| \\
|\mathbf{slam} a. e| &= \mathbf{slam} a. |e| \\
|\mathbf{sapp}(e, s)| &= \mathbf{sapp}(|e|, |s|)
\end{aligned}$$

Fig. 15. The proof-erasure function $|\cdot|$ on dynamic terms

- For each sort σ , the existential quantifier \exists_σ of the sort $(\sigma \rightarrow \text{bool}) \Rightarrow \text{bool}$ corresponds to the existential quantifier \exists_σ of the sort $(\sigma \rightarrow \text{prop}) \Rightarrow \text{prop}$.

For every static term s , $|s|$ is the static term obtained from replacing in s each σ with $|\sigma|$ and each scx with $|scx|$.

Proposition 4.1

Assume that $\Sigma \vdash s : \sigma$ is derivable. Then $|\Sigma| \vdash |s| : |\sigma|$ is also derivable.

Proof

By induction on the sorting derivation of $\Sigma \vdash s : \sigma$. □

For a sequence \vec{B} of static boolean terms, $|\vec{B}|$ is the sequence obtained from applying $|\cdot|$ to each B in \vec{B} .

There are two functions $|\cdot|_p$ and $|\cdot|_t$ for mapping a given dynamic variable context Δ to a sequence of boolean terms and a dynamic variable context, respectively:

- $|\Delta|_p$ is a sequence of boolean terms \vec{B} such that each B in \vec{B} is $|P|$ for some $a : P$ declared in Σ .
- $|\Delta|_t$ is a dynamic variable context such each declaration in it is of the form $a : |T|$ for some $a : T$ declared in Σ .

The proof-erasure function on dynamic terms is defined in Figure 15. Clearly, given a dynamic term e in ATS_{pf} , $|e|$ is a dynamic term in ATS_0 if it is defined.

As the proof-erasure of \leq_{pr} is chosen to be the boolean implication function, it needs to be assumed that $\Sigma; \vec{B} \vdash P_1 \leq_{pr} P_2$ implies $|\Sigma|; |\vec{B}| \vdash |P_1| \supset |P_2|$

Lemma 4.1 (Constraint Internalization)

Assume that the typing judgment $\Sigma; \vec{B}; \Delta \vdash e : P$ is derivable in ATS_{pf} . Then the constraint $|\Sigma|; |\vec{B}|, |\Delta|_p \models |P|$ holds.

Proof

By structural induction on the typing derivation \mathcal{D} of $\Sigma; \vec{B}; \Delta \vdash e : P$. Note that the typing rule **(ty-sub-p)** is handled by the assumption that $\Sigma; \vec{B} \vdash P_1 \leq_{pr} P_2$ implies $|\Sigma|; |\vec{B}| \vdash |P_1| \supset |P_2|$ for any props P_1 and P_2 .

- Assume that the last applied rule in \mathcal{D} is **(ty-tup-pp)**:

$$\frac{\mathcal{D}_1 :: \Sigma; \vec{B}; \Delta \vdash e_1 : P_1 \quad \mathcal{D}_2 :: \Sigma; \vec{B}; \Delta \vdash e_2 : P_2}{\Sigma; \vec{B}; \Delta \vdash \langle e_1, e_2 \rangle_{pp} : P_1 * P_2} \text{ (ty-tup-pp)}$$

where $P = P_1 * P_2$. By induction hypothesis on \mathcal{D}_1 , $|\Sigma|; |\vec{B}|, |\Delta|_p \models |P_1|$ holds. By induction hypothesis on \mathcal{D}_2 , $|\Sigma|; |\vec{B}|, |\Delta|_p \models |P_2|$ holds. Note that $|P| = |P_1 * P_2| = |P_1| \wedge |P_2|$, where \wedge stands for the boolean conjunction. Therefore, $|\Sigma|; |\vec{B}|, |\Delta|_p \models |P|$ holds.

- Assume that the last applied rule in \mathcal{D} is either **(ty-fst-pp)** or **(ty-snd-pp)**. This case immediately follows from the fact that $|P_1 * P_2| = |P_1| \wedge |P_2|$ for any props P_1 and P_2 , where \wedge stands for the boolean conjunction
- Assume that the last applied rule in \mathcal{D} is **(ty-lam-pp)**:

$$\frac{\mathcal{D}_1 :: \Sigma; \vec{B}; \Delta, x_1 : P_1 \vdash e_2 : P_2}{\Sigma; \vec{B}; \Delta \vdash \text{lam}_{pp} x_1. e_2 : P_1 \rightarrow P_2} \text{ (ty-lam-pp)}$$

where $P = P_1 \rightarrow P_2$. By induction hypothesis on \mathcal{D}_1 , $|\Sigma|; |\vec{B}|, |\Delta|_p, |P_1| \models |P_2|$ holds. By the regularity rule **(reg-cut)**, $|\Sigma|; |\vec{B}|, |\Delta|_p \models |P_2|$ holds whenever $|\Sigma|; |\vec{B}|, |\Delta|_p \models |P_1|$ holds. Therefore, $|\Sigma|; |\vec{B}|, |\Delta|_p \models |P_1| \supset |P_2|$ holds, where \supset stands for the boolean implication. Note that $|P| = |P_1| \supset |P_2|$, and this case concludes.

- Assume that the last applied rule in \mathcal{D} is **(ty-app-pp)**:

$$\frac{\mathcal{D}_1 :: \Sigma; \vec{B}; \Delta \vdash e_1 : P_1 \rightarrow P_2 \quad \mathcal{D}_2 :: \Sigma; \vec{B}; \Delta \vdash e_2 : P_1}{\Sigma; \vec{B}; \Delta \vdash \text{app}_{pp}(e_1, e_2) : P_2} \text{ (ty-app-pp)}$$

where $P = P_2$. By induction hypothesis on \mathcal{D}_1 , $|\Sigma|; |\vec{B}|, |\Delta|_p \models |P_1| \supset |P_2|$ holds, where \supset stands for the boolean implication. By induction hypothesis on \mathcal{D}_2 , the constraint $|\Sigma|; |\vec{B}|, |\Delta|_p \models |P_1|$ holds. Therefore, the constraint $|\Sigma|; |\vec{B}|, |\Delta|_p \models |P_2|$ also holds.

The rest of the cases can be handled similarly. \square

Note that a proof in ATS_{pf} can be non-constructive as it is not expected for the proof to have any computational meaning. In particular, one can extend the proof construction in ATS_{pf} with any kind of reasoning based on classical logic (e.g., double negation elimination).

If a c-type CT assigned to a dynamic (proof) constant is of the form $\forall \Sigma. \vec{B} \supset (\vec{P}) \Rightarrow P_0$, then it is assumed that the following constraint holds in ATS_0 :

$$\emptyset; \emptyset \models \forall |\Sigma|. |\vec{B}| \supset (|\vec{P}| \supset |P_0|)$$

For instance, the c-types assigned to `fact_p_bas` and `fact_p_ind` imply the validity of the following constraints:

$$\begin{aligned} \emptyset; \emptyset &\vdash \text{fact_b}(0, 1) \\ \emptyset; \emptyset &\vdash \forall n : \text{int}. \forall r : \text{int}. (n \geq 0 \wedge \text{fact_b}(n, r) \supset \text{fact_b}(n+1, (n+1) \cdot r)) \end{aligned}$$

which are encoded directly into the c-types assigned to `fact_b_bas` and `fact_b_ind`.

If a c-type CT is of the form $\forall \Sigma. \vec{B} \supset (\vec{P}, T_1, \dots, T_n) \Rightarrow T_0$, then $|CT|$ is defined as follows:

$$\forall |\Sigma|. |\vec{B}| \supset (|\vec{P}| \supset ((|T_1|, \dots, |T_n|) \Rightarrow |T_0|))$$

If a dynamic constant dex is assigned the c-type CT in ATS_{pf} , then it is assumed to be of the c-type $|CT|$ in ATS_0 .

Theorem 4.1

Assume that $\Sigma; \vec{B}; \Delta \vdash e : T$ is derivable in ATS_{pf} . Then $|\Sigma|; |\vec{B}|, |\Delta|_p; |\Delta|_t \vdash |e| : |T|$ is derivable in ATS_0 .

Proof

By structural induction on the typing derivation \mathcal{D} of $\Sigma; \vec{B}; \Delta \vdash e : T$.

- Assume that the last applied rule in \mathcal{D} is **(ty-*elim-pt)**:

$$\frac{\mathcal{D}_1 :: \Sigma; \vec{B}; \Delta \vdash e_{12} : P_1 * T_2 \quad \mathcal{D}_2 :: \Sigma; \vec{B}; \Delta, x_1 : P_1, x_2 : T_2 \vdash e_0 : T_0}{\Sigma; \vec{B}; \Delta \vdash \mathbf{let} \langle x_1, x_2 \rangle_{pt} = e_{12} \mathbf{in} e_0 : T_0} \text{ (ty-*elim-pt)}$$

where e is $\mathbf{let} \langle x_1, x_2 \rangle_{pt} = e_{12} \mathbf{in} e_0$ and $T = T_0$. By induction hypothesis on \mathcal{D}_1 , there exists the following derivation in ATS_0 :

$$\mathcal{D}'_1 :: |\Sigma|; |\vec{B}|, |\Delta|_p; |\Delta|_t \vdash |e_{12}| : |P_1| \wedge |T_2|$$

By induction hypothesis on \mathcal{D}_2 , there exists the following derivation in ATS_0 :

$$\mathcal{D}'_2 :: |\Sigma|; |\vec{B}|, |\Delta|_p, |P_1|; |\Delta|_t, x_2 : |T_2| \vdash |e_0| : |T_0|$$

Applying the rule **(ty- \wedge -elim)** to \mathcal{D}'_1 and \mathcal{D}'_2 yields the following derivation:

$$\mathcal{D}' :: |\Sigma|; |\vec{B}|, |\Delta|_p; |\Delta|_t \vdash \mathbf{let} \wedge(x_2) = |e_{12}| \mathbf{in} |e_0| : |T_0|$$

Note that $|e|$ equals $\mathbf{let} \wedge(x_2) = |e_{12}| \mathbf{in} |e_0|$, and the case concludes.

- Assume that the last applied rule in \mathcal{D} is **(ty-app-tp)**:

$$\frac{\mathcal{D}_1 :: \Sigma; \vec{B}; \Delta \vdash e_1 : P_1 \rightarrow T_2 \quad \mathcal{D}_2 :: \Sigma; \vec{B}; \Delta \vdash e_2 : P_1}{\Sigma; \vec{B}; \Delta \vdash \mathbf{app}_{tp}(e_1, e_2) : T_2} \text{ (ty-app-tp)}$$

where e is $\mathbf{app}_{tp}(e_1, e_2)$ and $T = T_2$. By induction hypothesis on \mathcal{D}_1 , there exists the following derivation in ATS_0 :

$$\mathcal{D}'_1 :: |\Sigma|; |\vec{B}|, |\Delta|_p; |\Delta|_t \vdash |e_1| : |P_1| \supset |T_2|$$

Applying Lemma 4.1 to \mathcal{D}_2 yields that the constraint $|\Sigma|; |\vec{B}|, |\Delta|_p; |\Delta|_t \vdash |P_1|$ is valid.

Applying the rule **(ty- \supset -elim)** to \mathcal{D}'_1 and the valid constraint yields the following derivation:

$$|\Sigma|; |\vec{B}|, |\Delta|_p; |\Delta|_t \vdash \supset^-(|e_1|) : |T_2|$$

Note that $|e|$ equals $\supset^-(|e_1|)$, and the case concludes.

The rest of the cases can be handled similarly. \square

By Theorem 4.1, the proof-erasure of a program is well-typed in ATS_0 if the program itself is well-typed in ATS_{pf} . In other words, Theorem 4.1 justifies PwTP in ATS_{pf} as an approach to internalizing constraint-solving through explicit proof-construction.

5 Related Work and Conclusion

Constructive type theory, which was originally proposed by Martin-Löf for the purpose of establishing a foundation for mathematics, requires pure reasoning on programs. Generalizing as well as extending Martin-Löf's work, the framework Pure Type System (**PTS**) offers a simple and general approach to designing and formalizing type systems. However, type equality depends on program equality in the presence of dependent types, making it highly challenging to accommodate effectful programming features as these features often greatly complicate the definition of program equality (Constable & Smith, 1987; Mendler, 1987; Honsell *et al.*, 1995; Hayashi & Nakano, 1988).

The framework *Applied Type System* (**ATS**) (Xi, 2004) introduces a complete separation between statics, where types are formed and reasoned about, and dynamics, where programs are constructed and evaluated, thus eliminating by design the need for pure reasoning on programs in the presence of dependent types. The development of **ATS** primarily unifies and also extends the previous studies on both Dependent ML (DML) (Xi & Pfenning, 1999; Xi, 2007) and guarded recursive datatypes (Xi *et al.*, 2003). DML enriches ML with a restricted form of dependent datatypes, allowing for specification and inference of significantly more precise type information (when compared to ML), and guarded recursive datatypes can be thought of as an impredicative form of dependent types in which type indexes are themselves types. Given the similarity between these two forms of types, it is only natural to seek a unified presentation for them. Indeed, both DML-style dependent types and guarded recursive datatypes are accommodated in **ATS**.

In terms of theorem-proving, there is a fundamental difference between **ATS** and various theorem-proving systems such as NuPrl (Constable *et al.*, 1986) (based on Martin-Löf's constructive type theory) and Coq (Dowek *et al.*, 1993) (based on the calculus of construction (Coquand & Huet, 1988)). In **ATS**, proof construction is solely meant for constraint simplification and proofs are not expected to contain any computational meaning. On the other hand, proofs in NuPrl and Coq are required to be constructive as they are meant for supporting program extraction.

The theme of combining programming with theorem-proving is also present in the programming language Ωemga (Sheard, 2004). The type system of Ωemga is largely built on top of a notion called *equality constrained types* (a.k.a. phantom types (Cheney & Hinze, 2003)), which are closely related to the notion of guarded recursive datatypes (Xi *et al.*, 2003). In Ωemga , there seems no strict separation between programs and proofs. In particular, proofs need to be constructed at run-time. In addition, an approach to simulating dependent types through the use of type classes in Haskell is given in (McBride, 2002), which is casually related to proof construction in the design of **ATS**. Please also see (Chen *et al.*, 2004) for a critique on the practicality of simulating dependent types in Haskell.

In summary, a framework **ATS** is presented in this paper to facilitate the design and formalization of type systems to support practical programming. With a complete separation between statics and dynamics, **ATS** removes by design the need for pure reasoning on programs in the presence of dependent types. Additionally, **ATS** allows programming and theorem-proving to be combined in a syntactically intertwined manner, providing the programmer with an approach to internalizing constraint-solving through explicit proof-construction. As a minimalist formulation of **ATS**, ATS_0 is first presented and its type-

soundness formally established. Subsequently, ATS_0 is extended to ATS_{pf} so as to support programming with theorem-proving, and the correctness of this extension is proven based on a translation often referred to as proof-erasure, which turns each well-typed program in ATS_{pf} into a corresponding well-typed program in ATS_0 of the same dynamic semantics.

References

- Barendregt, Hendrik Pieter. (1992). Lambda Calculi with Types. *Pages 117–441 of: Abramsky, S., Gabbay, Dov M., & Maibaum, T.S.E. (eds), Handbook of Logic in Computer Science*, vol. II. Oxford: Clarendon Press.
- Chen, Chiyan, & Xi, Hongwei. 2005 (September). Combining Programming with Theorem Proving. *Pages 66–77 of: Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*.
- Chen, Chiyan, Zhu, Dengping, & Xi, Hongwei. (2004). Implementing Cut Elimination: A Case Study of Simulating Dependent Types in Haskell. *Proceedings of the 6th International Symposium on Practical Aspects of Declarative Languages*. Dallas, TX: Springer-Verlag LNCS vol. 3057.
- Cheney, James, & Hinze, Ralf. (2003). *Phantom Types*. Technical Report CUCIS-TR2003-1901. Cornell University.
- Constable, Robert L., & Smith, Scott Fraser. 1987 (June). Partial objects in constructive type theory. *Pages 183–193 of: Proceedings of Symposium on Logic in Computer Science*.
- Constable, Robert L., et al. . (1986). *Implementing mathematics with the NuPrl proof development system*. Englewood Cliffs, New Jersey: Prentice-Hall.
- Coquand, Thierry, & Huet, Gérard. (1988). The calculus of constructions. *Information and computation*, **76**(2–3), 95–120.
- de Moura, Leonardo Mendonça, & Bjørner, Nikolaj. (2008). Z3: an efficient SMT solver. *Pages 337–340 of: Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29–April 6, 2008. Proceedings*.
- Dowek, Gilles, Felty, Amy, Herbelin, Hugo, Huet, Gérard, Murthy, Chet, Parent, Catherine, Paulin-Mohring, Christine, & Werner, Benjamin. (1993). *The Coq proof assistant user's guide*. Rapport Technique 154. INRIA, Rocquencourt, France. Version 5.8.
- Girard, Jean-Yves. (1986). The System F of variable types, fifteen years later. *Theoretical computer science*, **45**(2), 159–192.
- Hayashi, Susumu, & Nakano, Hiroshi. (1988). *PX: A computational logic*. The MIT Press.
- Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Communications of the acm*, **12**(10), 576–580 and 583.
- Honsell, Furio, Mason, Ian A., Smith, Scott, & Talcott, Carolyn. (1995). A variable typed logic of effects. *Information and computation*, **119**(1), 55–90.
- Jones, Mark P. (1994). *Qualified types: Theory and practice*. The Edinburgh Building, Cambridge CB2 2RU, UK: Cambridge University Press.
- McBride, Conor. (2002). Faking It. *Journal of functional programming*, **12**(4 & 5), 375–392.
- Mendler, N.P. 1987 (June). Recursive types and type constraints in second-order lambda calculus. *Pages 30–36 of: Proceedings of Symposium on Logic in Computer Science*. The Computer Society of the IEEE, Ithaca, New York.
- Milner, Robin, Tofte, Mads, Harper, Robert W., & MacQueen, D. (1997). *The definition of standard ml (revised)*. Cambridge, Massachusetts: MIT Press.
- Reynolds, John C. (1972). Definitional interpreters for higher-order programming languages. *Pages 717–740 of: Proceedings of the ACM Annual Conference*.
- Reynolds, John C. (1998). *Theories of programming languages*. Cambridge University Press.

- Sheard, Tim. (2004). Languages of the future. *Proceedings of the Onward! Track of Objected-Oriented Programming Systems, Languages, Applications (OOPSLA)*. Vancouver, BC: ACM Press.
- Xi, Hongwei. (2001). *Dependent ML*. Available at:
<http://www.cs.bu.edu/~hwxi/DML/DML.html>.
- Xi, Hongwei. (2004). Applied Type System (extended abstract). *Pages 394–408 of: post-workshop Proceedings of TYPES 2003*. Springer-Verlag LNCS 3085.
- Xi, Hongwei. (2007). Dependent ML: An approach to practical programming with dependent types. *Journal of functional programming*, **17**(2), 215–286.
- Xi, Hongwei. (2008a). ATS/LF: a type system for constructing proofs as total functional programs. Benz Müller, Christoph, Brown, Chad, Siekmann, Jörg, & Statman, Rick (eds), *Festschrift in Honour of Peter B. Andrews on his 70th Birthday*. Studies in Logic and the Foundations of Mathematics. IFCoLog.
- Xi, Hongwei. (2008b). *The ATS Programming Language System*. Available at:
<http://www.ats-lang.org/>.
- Xi, Hongwei, & Pfenning, Frank. (1999). Dependent Types in Practical Programming. *Pages 214–227 of: Proceedings of 26th ACM SIGPLAN Symposium on Principles of Programming Languages*. San Antonio, Texas: ACM press.
- Xi, Hongwei, Chen, Chiyan, & Chen, Gang. (2003). Guarded Recursive Datatype Constructors. *Pages 224–235 of: Proceedings of the 30th ACM SIGPLAN Symposium on Principles of Programming Languages*. New Orleans, LA: ACM press.